HA & BC

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#57 APRIL 1995

SPECIAL BONUS SECTION: HOME AUTOMATION & BUILDING CONTROL

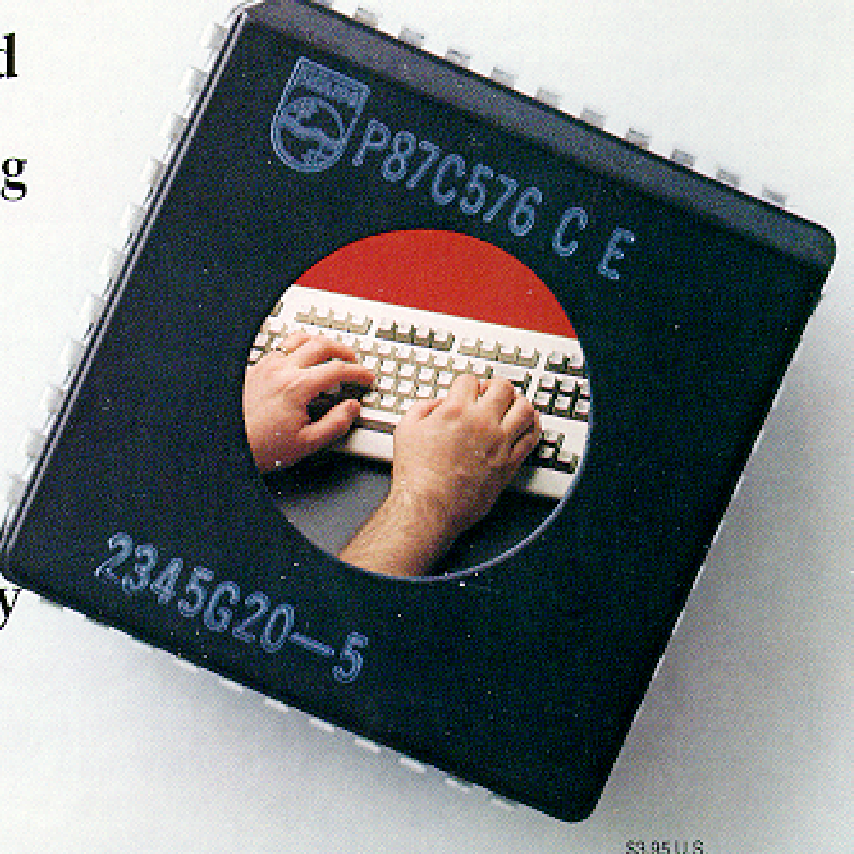
EMBEDDED PROGRAMMING

PC/104 Explained

C++ Programming Primer

Characterizing Processor Performance

Sensing Humidity and Acceleration



$3.95 U.S.
$4.95 Canada

EDITOR'S INK

A Reaffirmation

having just finished the Version 3.0 release of the HCS/SpectraSense software, I've reaffirmed a belief I've held for many years now. I'd rather develop real-time code for an embedded controller using assembly language than write application code for a desktop machine using a high-level language.

I know I'm in the minority for holding such a position, but I'm sure many *INK* readers agree with me.

The HCS/SpectraSense suite consists of real-time control code running on the Supervisory Controller with the compiler and host code running on a PC. The former is written in 100% HD64180 (Z180) assembly language while the latter is in Borland C.

For me, the embedded code is always much more enjoyable to write and debug. It works much closer to my expectations and tends to be more reliable. Given the choice between reliability in code that runs 24 hours a day versus code that is fired up perhaps every few weeks, I'll take the first one.

I'm sure a lot of my preference has a lot to do with the fact that while working on the controller code, I have more control. I get to deal with inanimate objects that don't mind binary and generally hand me predictable data. Having to deal with the unpredictability of human behavior though, takes programming to another plateau.

The other issue is one Ed has dealt with in his "Firmware Furnace" columns: proximity to the guts of the hardware. If I can twiddle the bits directly from my program, I can write device drivers that behave as I want. I know where the code has been and I know where it's going. I become afraid of heights when there are too many levels of software under me.

Some "embedded programming" does involve the use of high-level languages, and I've even written about the virtues of doing so. Many of the articles in this "Embedded Programming" issue don't necessarily limit themselves to assembly language, so I'll let you decide.

In the meantime, I'll go back to thinking in hex.

Ken

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

FOUNDER/EDITORIAL DIRECTOR
Steve Ciarcia

EDITOR-IN-CHIEF
Ken Davidson

TECHNICAL EDITOR
Janice Marinelli

ENGINEERING STAFF
Jeff Bachiochi & Ed Nisley

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITOR
John Dybowski

NEW PRODUCTS EDITOR
Harv Weiner

ART DIRECTOR
Lisa Ferry

PRODUCTION STAFF
John Gorsky
James Soussounis

CONTRIBUTORS:
Jon Elson
Tim McDonough
Frank Kuechmann
Pellervo Kaskinen

PUBLISHER
Daniel Rodrigues

PUBLISHER'S ASSISTANT
Sue Hodge

CIRCULATION COORDINATOR
Rose Mansella

CIRCULATION ASSISTANT
Barbara Maleski

CIRCULATION CONSULTANT
Gregory Spitzfaden

BUSINESS MANAGER
Jeannette Walters

ADVERTISING COORDINATOR
Dan Gorsky

CIRCUIT CELLAR INK, THE COMPUTER APPLICATIONS JOURNAL (ISSN 0896-8985) is published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (203) 875-2751. Second class postage paid at Vernon, CT and additional offices. One-year (12 issues) subscription rate U.S.A. and possessions $21.95, Canada/Mexico $31.95, all other countries $49.95. All subscription orders payable in U.S. funds only, via international postal money order or check drawn on U.S. bank. Direct subscription orders and subscription related questions to Circuit Cellar INK Subscriptions, P.O. Box 698, Holmes, PA 19043-9613 or call (800) 269-6301.
POSTMASTER: Please send address changes to Circuit Cellar INK, Circulation Dept., P.O. Box 698, Holmes, PA 19043-9613.

Cover photography by Barbara Swenson
PRINTED IN THE UNITED STATES

HAJAR ASSOCIATES **NATIONAL ADVERTISING REPRESENTATIVES**

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

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

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

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

Circuit Cellar BBS—24 Hrs. 300/1200/2400/9600/14.4k bps, 8 bits, no parity, 1 stop bit, (203) 871-1988; 2400/9600 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*.

Entire contents copyright © 1995 by Circuit Cellar Incorporated. All rights reserved. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

12 A C++ Programming Tutorial
by Mike Podanoffsky

22 Characterizing Processor Performance
by Rick Naro

26 Designing with PC/104
by Rick Lehrbaum

36 An LCD and Keypad Module for the SPI
by Brian Millier

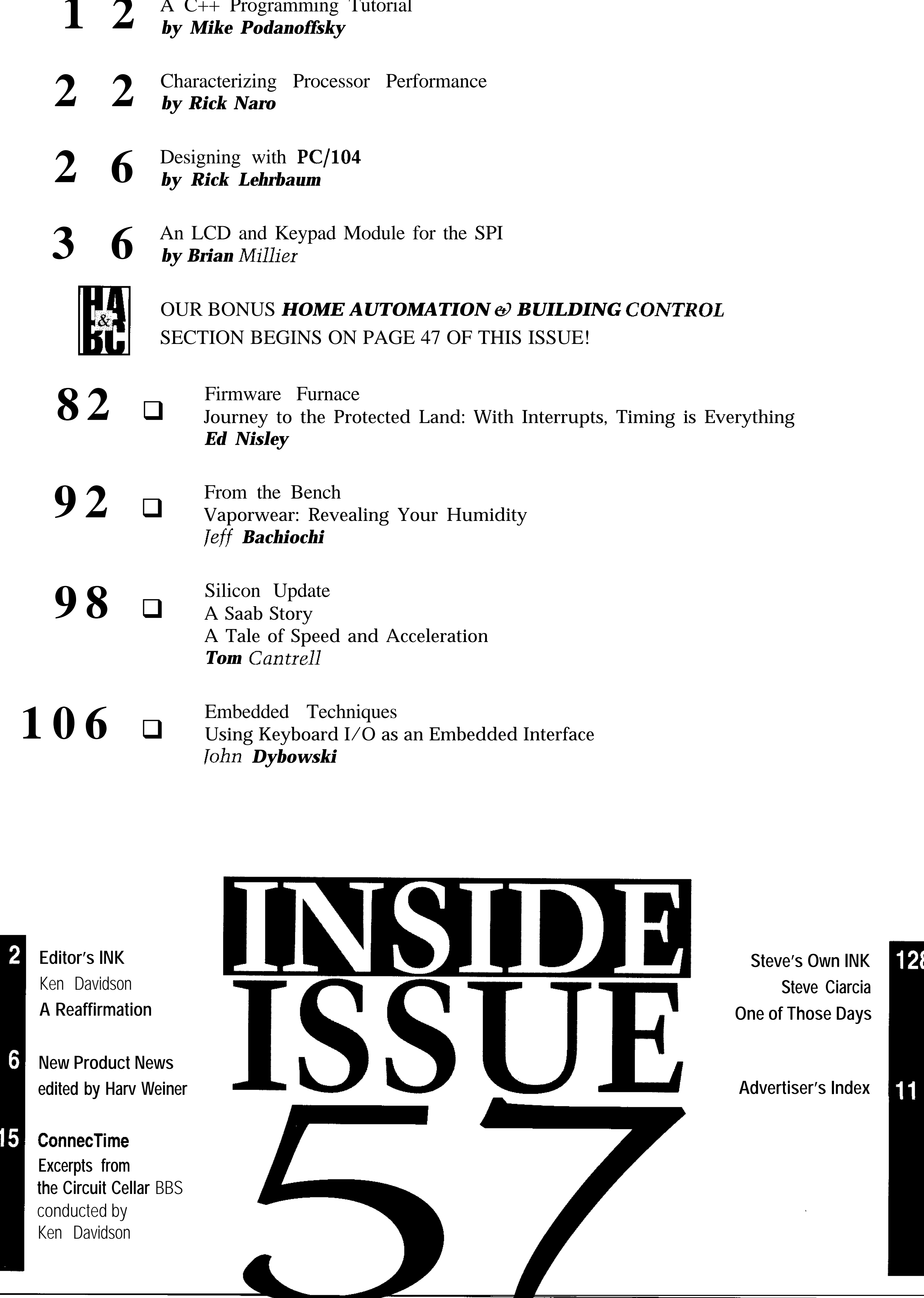
OUR BONUS ***HOME AUTOMATION & BUILDING CONTROL***
SECTION BEGINS ON PAGE 47 OF THIS ISSUE!

82 ❑ Firmware Furnace
Journey to the Protected Land: With Interrupts, Timing is Everything
Ed Nisley

92 ❑ From the Bench
Vaporwear: Revealing Your Humidity
Jeff Bachiochi

98 ❑ Silicon Update
A Saab Story
A Tale of Speed and Acceleration
Tom Cantrell

106 ❑ Embedded Techniques
Using Keyboard I/O as an Embedded Interface
John Dybowski



INSIDE ISSUE 57

2 Editor's INK
Ken Davidson
A Reaffirmation

6 New Product News
edited by Harv Weiner

115 ConnecTime
Excerpts from the Circuit Cellar BBS
conducted by Ken Davidson

128 Steve's Own INK
Steve Ciarcia
One of Those Days

11 Advertiser's Index

NEW PRODUCT NEWS

Edited by Harv Weiner

PORTABLE DATA ACQUISITION SYSTEM

Industrial Computer Source has announced **DAQBOOK/100**, a high-speed, multifunction, data-acquisition subsystem for notebook PCs. Power to the unit may be supplied by a number of sources: a 12-V car battery, 120-VAC line power, or an optional rechargeable NiCd battery.

The unit combines the functionality of several plug-in analog and digital data-acquisition boards in an external module the size of a notebook PC. Attaching directly to a portable PC's parallel port, DAQBOOK/100 provides 16 single-ended or 8 differential inputs with a bidirectional data-transfer rate of up to 170 kBps. Acquired data can be stored real time in the PC's memory and hard drive.

Software includes full-featured DOS and Windows drivers for C, BASIC, and Visual Basic. As well, DaqView, a Windows graphics application, enables the user to set up an application to acquire and save data directly to disk or to seamlessly transmit it to other Window applications. VISUALAB is another software option, offering a set of DLLs to extend the capabilities of Visual Basic and Snap-Master for Windows. This advanced software package integrates data acquisition, high-speed data streaming to disk, data retrieval, and analysis.

In addition to portable testing, DAQBOOK/100 is ideal for remote data-collection applications such as automotive and aviation in-vehicle testing. An expansion chassis (Model **DBK10**) provides connection for multiplexers, thermocouple cards, a strain-gauge interface, and more digital channels.

DAQBOOK/100 sells for $1295 and includes an AC adapter, parallel-port cable, DOS and Windows drivers, and DaqView software. The expansion chassis sells for $150.

Industrial Computer Source
3950 Barnes Canyon Rd.
San Diego, CA 92121
(619) 677-0877 • Fax: (619) 677-0898

#500

SERIAL EPROM EMULATOR

Softec Microsystems introduces a serial EPROM emulator that doesn't require removal of the system microprocessor. Unlike in-circuit emulators that replace the microprocessor with a pod, the EMUR7 replaces and emulates the system EPROM. Designers then use their own development tools [assembler, compiler, and linker) to execute object code on the test board as if a new EPROM had been programmed. This eliminates the need to remove, erase, reprogram, and reinsert the EPROM.

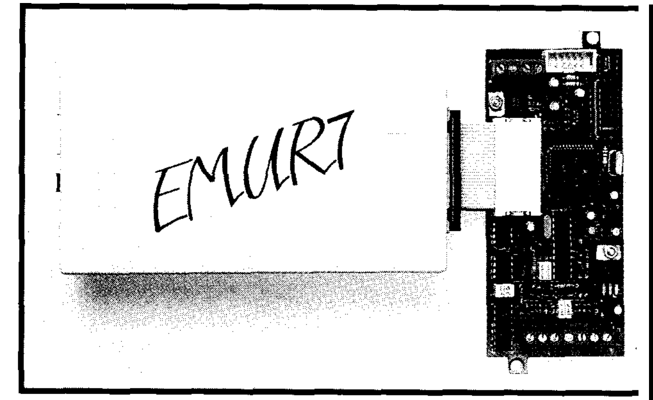
This emulation approach has several advantages. The emulator is truly universal and the designer may choose the family or processor satisfying individual application requirements without buying expensive in-circuit emulators for different processors. Another advantage is that the test circuit works under normal operating conditions-the real microprocessor guarantees that all electrical and time parameters comply to spec.

EMUR7 supports EPROM devices up to 8 Mb [the basic version includes a l-Mb emulation RAM). In addition to loading the object code at the rate of 115,200 bps, receives the traditional binary, Motorola-S, and Intel hex formats. The EMUR7 connects to a PC-compatible computer through the serial port and does not need a power supply. The unit can be connected to a portable PC when electric power is not available.

The EMUR7 includes a friendly user interface with working selections and options clearly and efficiently displayed. The unit offers full mouse support, a context-sensitive help feature, and a 43-line video mode. The integrated ROM editor lets users modify the emulated code.

Softec Microsystems
33082 Azzano Decimo (PN
Italy
+39 434 640113
Fax: t39 434 631598

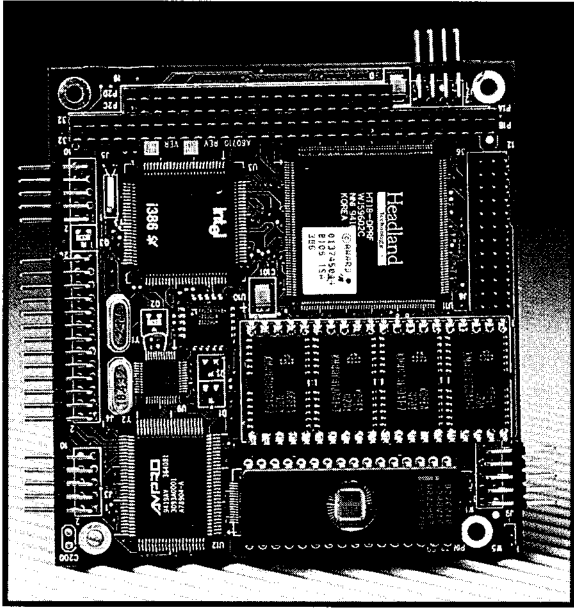
#501



NEW PRODUCT NEWS

'386SX MODULE

Ampro Computers is offering a new '386SX PC/104 CPU module. The **CoreModule/386-II** is based on Intel's 25-MHz '386SX CPU and includes two serial ports, up to 16 MB of DRAM as well as onboard NVRAM or a flash memory, solid-state disk. CoreModule/386-II complies with the newly adopted PC/104 (V. 2) standard. Typical applications include medical instruments, vehicular data acquisition and control systems, and portable test equipment.



The CoreModule/386-II contains the equivalent of a complete PC/AT motherboard and several expansion cards. Onboard I/O functions include two RS-232 serial ports, a bidirectional parallel port, as well as standard keyboard and speaker interfaces. An onboard, bootable solid-state disk assures reliable operation in harsh operating environments. Watchdog timer and power monitor functions are also included to ensure maximum system integrity in critical applications. The module operates with approximately 3 W (active mode) of power and is designed for use in extended temperature environments of 0-70°C.

The CoreModule/386-II sells for $359 in quantity.

Ampro Computers, Inc.
990 **Almanor** Ave.
Sunnyvale, CA 94086
(408) 522-2100
Fax: (408) 720-l 305

#502

don't like windows?

don't care for r&d?

sorry.

Then we just can't help you. But if you're looking for a high-capacity, user-friendly EDA system, we've got just what you need. Say "hi" to EDWin, your new companion in Electronics Design. EDWin features seamless integration between modules, so you can finally kiss the tedious concept of front- and back annotation goodbye. EDWin gives you all the tools you'll need, and is so user-friendly you can even compile your own custom toolboxes. So easy to learn, you'll be up and running in minutes, EDWin also features nice pricing, starting at just $495.

Make your appointment with us today for the EDWin evaluation package. Welcome.

Vision EDA Corp.
995 E Baseline Rd. Ste 2166,
Tempe, Arizona 85283-l 336
Phone: l-800-EDA-4-YOU, or (602) 730 8900
Fax: (602) 730 8927

ELECTRONICS DESIGN FOR WINDOWS

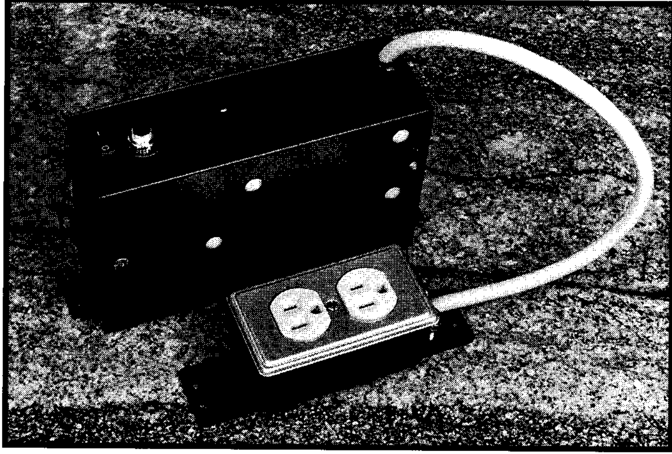
EDWIN

EDWin is a trademark of Norlinvest Ltd. Windows is a trademark of Microsoft Corp.

#103

NEW PRODUCT NEWS

SERIES-MODE SURGE PROTECTOR



Zero Surge announces a customizable OEM version of its **series-mode surge protector**. Sensitive low-energy circuits used in today's microelectronics are extremely vulnerable to spikes and surges in power lines.

Series-mode surge protectors reduce the intensity of the surge energy, storing the energy in capacitors and then draining it slowly onto the neutral wire without harm to motherboards and other interconnected circuitry.

Most point-of-use surge suppressers use inexpensive MOVs as their primary suppression components. These components wear out a little with each surge above a modest threshold, diminishing their ability to protect over time.

In contrast, Zero Surge protectors do not degrade even in severe surge environments and can withstand hundreds of maximum-rated surge pulses of 6,000 V, which is more than is expected in 10 years of service.

Zero Surge protectors offer power-line protection that provides low let-through voltage (under 200 V), does not use the safety ground as a surge sink, reduces the intensity of all surges, and intercepts all surge frequencies including internally generated high-frequency surges. The Zero Surge OEM surge-protection device has a master switch, thermal reset button, and 6′ line cord.

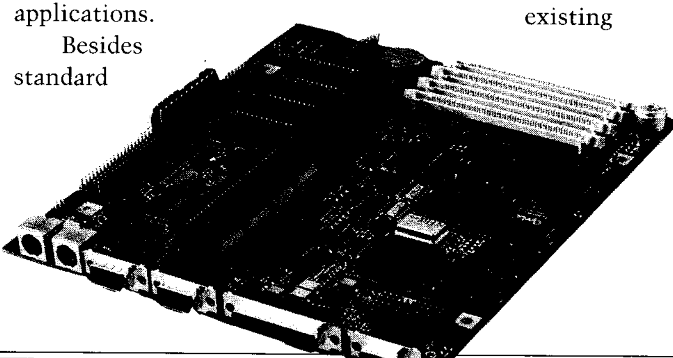
Zero Surge, Inc.
944 State Route 12
Frenchtown, NJ 08825
(908) 996-7700
Fax: (908) 996-7773 #503

SINGLE-BOARD COMPUTER

Dovatron International has introduced a PC/AT-compatible single-board computer. **PrimePlus** was specifically designed to meet the needs of product developers requiring full AT compatibility and to provide features commonly needed for embedded-control applications.

Besides standard processor and I/O functions, PrimePlus offers onboard ROM disk; SVGA graphics (which drives CRTs and LCDs); key-switch scanning; watchdog timer; two RS-232, RS-422, or RS-485 serial ports; plus power management and expansion capability for both standard ISA cards and Dovatron's miniature ESP Modules. Mezzanine-style expansion is available via existing ESP modules, which include flash, SCSI, Ethernet, analog-to-digital, digital-to-analog, PCMCIA, and more. PrimePlus was designed for industrial and/or machine control applications requiring rugged construction, expanded features, and power management. The board measures 8″ × 9″ and conforms to the LPM/LPX form-factor.

PrimePlus sells for **$571** in quantity.



Dovatron Products Division
4076 Specialty Pl.
Longmont, CO 80504
(303) 772-5933
Fax: (303) 772-3697

#504

PC/104 RESOURCE GUIDE

The PC/104 Consortium has announced the availability of the Sixth Edition (November 1994) of its popular **PC/104 Resource Guide**. Like its predecessors, the 170-page booklet is available free to engineers and companies developing embedded systems. It includes an overview of the PC/104 standard, a cross-reference of products, and product listings from the Consortium's member companies.

The PC/104 standard, which has also become the basis of a new IEEE draft standard (P996.1), defines a compact, self-stacking, modular form-factor for embedding IBM PC and PC/AT-compatible system functions within embedded microcomputer applications.

PC/104 Consortium
809 Cuesta Dr. B-175
Mountain View, CA 94040
(415) 903-8304
Fax: (415) 967-0995 #505

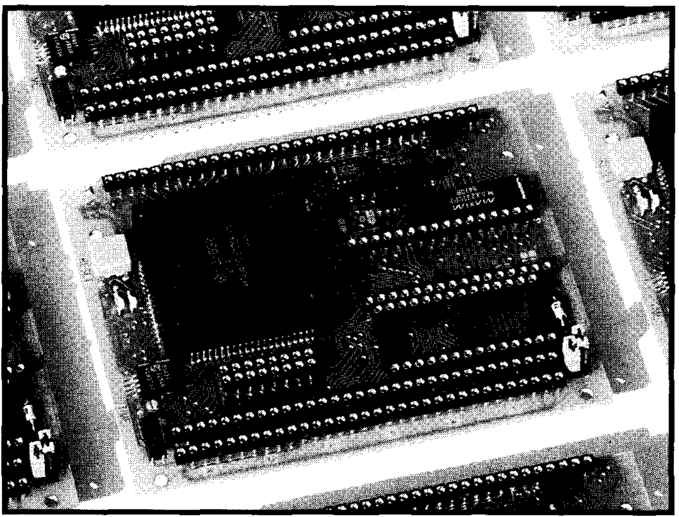
NEW PRODUCT NEWS

ANSI-FORTH DATALOGGER AND CONTROLLER

Saelig Company introduces a tailor-made data-collection system that can be read by a PC and features removable card memory. The **TDS2020** Data Logger Module adheres to the official Forth language definition. Forth is an easily-learned, high-level language ideal for fast control and well-suited to real-time embedded systems.

The TDS2020 is a 16-bit control computer featuring the Hitachi H8/532 CMOS microprocessor running at 20 MHz. It is available with **16** KB of ANSI-Forth kernel, a full symbolic assembler, 45 KB of program space, and up to **512 KB** of battery-backed RAM, EEPROM, or flash memory. In addition, a 40-MB miniature hard drive is available and 32 digital inputs may be monitored.

An onboard S-channel, 10-bit A/D converter and 3-channel, 8-bit D/A converter make the 4" x 3" board extremely versatile for data logging, robotics, or machine control. Up to 33 I/0 lines, two RS-232 serial lines, an I²C bus, real-time clock, and watchdog timer make an economical, versatile controller for a wide range of applications. Although small, it is packed with important features which make it easy to use in solving control problems. **A PC** library of ANSI-Forth software makes stepper-motor control, interrupt handling, real-time multitasking, data logging, serial I/O, keyboard, and LCD driving easy. When logging data in standby mode, it will run on 500 µA, so a 9-V battery lasts for a full year.



The TDS2020 sells for $499 for the starter pack, which includes a comprehensive manual and PC software.

The Saelig Company
1193 Moseley Rd.
Victor, NY 14564
(716) 425-3753
Fax: (716) 425-3835

#506

FOUR-PORT SERIAL BOARD

A PC-compatible serial card, featuring four serial ports in a single slot, is available from B&B Electronics. Each of the **3PXCC4A**'s ports can be independently configured for any I/O address and IRQ as well as RS-232, RS-422, or RS-485 data protocols, allowing it to fit any serial application.

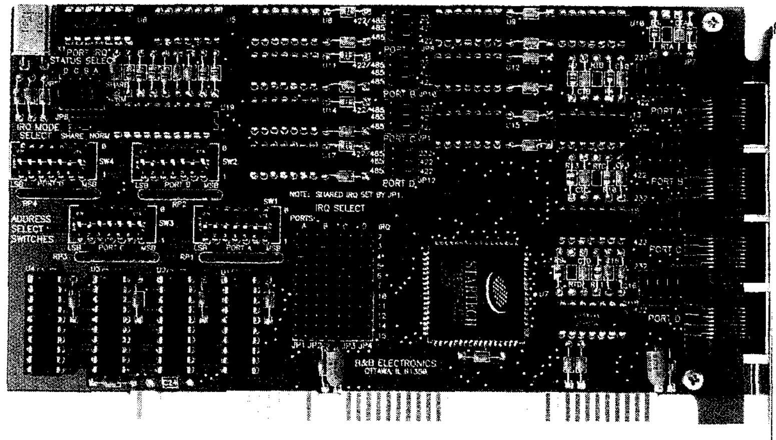
TD, RD, RTS, CTS, DSR, DCD, and DTR port lines are supported by the RS-232 mode with each port using a buffered, high-speed 16550A UART. The 3PXCC4A has interrupt-sharing capabilities and an interrupt status register to increase throughput in shared IRQ applications and the number of available interrupts in a system.

The card features eight-conductor RJ-45 connectors. Prewired adapter kits (Models MDB9 and MDB25) are available to convert the RJ-45 to DE9 or DB25 connectors.

The 3PXCC4A sells for $209.95 and the cable adapters are $10.95 each.

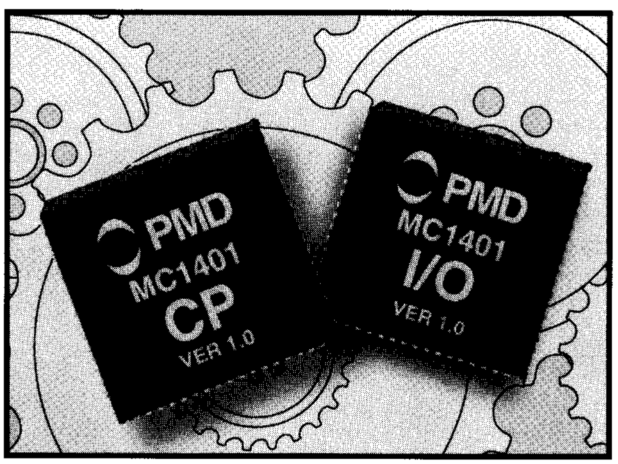
B&B Electronics Manufacturing Co.
P.O. Box 1040
Ottawa, IL 61350
(815) 434-0846
Fax: (815) **434-7094**
Internet: **catrqst@bb-elec.com**

#507



NEW PRODUCT NEWS

CHIPSET FEATURES ELECTRONIC GEARING



The MC1401 family of motion chipsets from PMD now supports up to two axes of servo-based electronic gearing. Electronic gearing can be used in numerous applications including robotics, medical automation, coil winding, and electronic camming.

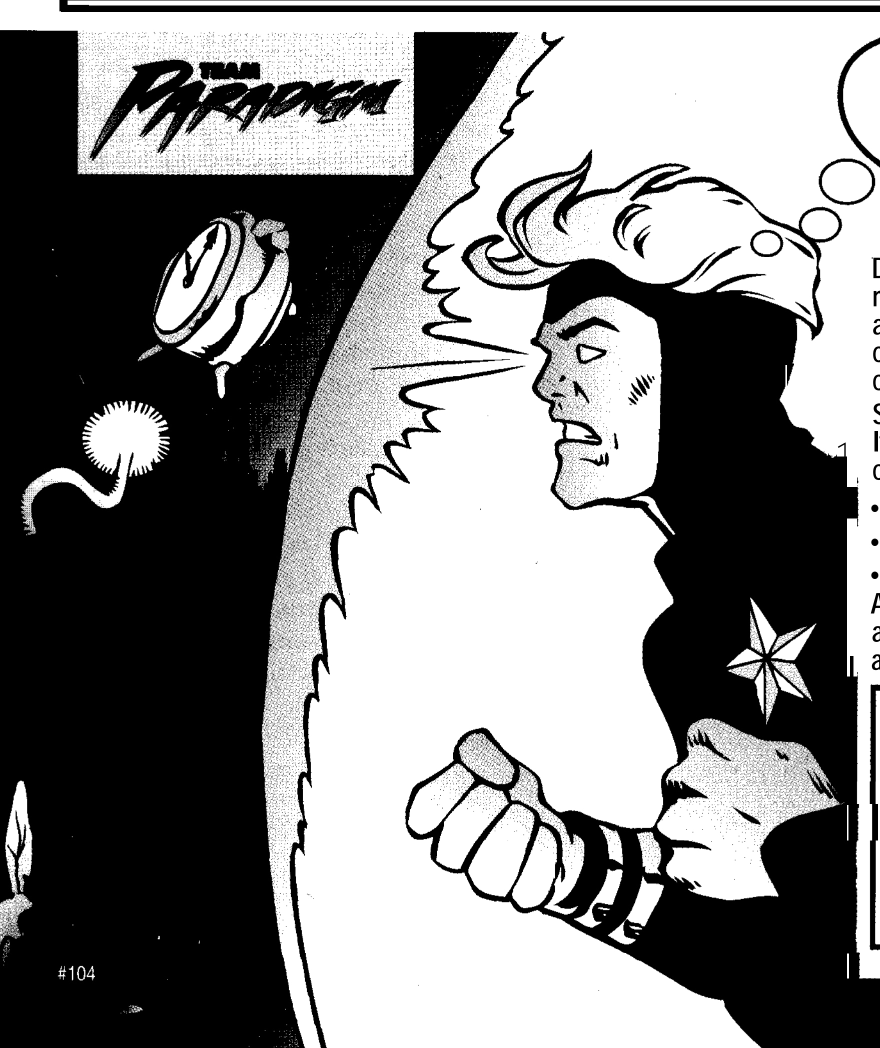
The required elements for electronic gearing are a master input-axis encoder and a servo-controlled slave axis. The position of the master axis is continuously monitored by the chipset and is used to drive the slave axis after going through a programmable 32-bit gear ratio. A special feature is that the master axis can itself be servo controlled, allowing the user to create tightly coupled multiaxis systems.

Other standard features of the chipset include three user-selectable profiling modes (S-curve, trapezoidal, and velocity contouring) along with high-resolution, 16-bit DAC output signals. The chipset provides closed-loop control using either a PID loop or a PI with velocity feed-forward feedback loop.

The MC 140 1 -series of motion chipsets are available in several versions, including one that supports incremental encoder input. Another supports absolute digital and resolver-based input and a third version supports sinusoidal commutation at up to 15 kHz.

The chipset is made up of two 68-pin PLCC packages and sells for $99 in quantity.

Performance Motion Devices, Inc.
11 Carriage Dr. • Chelmsford, MA 01824
(508) 256-l 913 • Fax: (508) 256-0206 **#508**



MUST PROTECT...!

Development tools alone aren't sufficient to make your '186 or V-Series design a success. Here at Paradigm, we have the tools, experience and commitment to see your design through to completion.

Start with Paradigm LOCATE and Paradigm DEBUG. If, along the way, you stumble or hit a brick wall, call Team Paradigm for help:

- Toll-free technical support
- 24-hour BBS support
- Paradigms customer newsletter

After all, life is tough enough without worrying about your development tools. Choose Paradigm and enjoy sleeping again at night.

PARADIGM 'Nuff said.

Proven Solutions for Embedded C/C++ Developers

1-800-537-5043

Paradigm Systems
3301 Country Club Road, Suite 2214, Endwell, NY 13760
(607) 748-5966 / FAX: (607) 748-5968
Internet: 73047.3031@ compuserve.com

©1995 Paradigm Systems, Inc. All rights reserved.

#104

FEATURES

12 A C++ Programming Tutorial

22 Characterizing Processor Performance

26 Designing with PC/104

36 An LCD and Keypad Module for the SPI

FEATURE ARTICLE

Mike Podanoffsky

A Ctt Programming Tutorial

Need help with the paradigm shift from a procedural C to an object-oriented C++? A step-by-step tutorial and lots of examples should help set you on the right track.

O his article should probably be entitled "C++ For Those Who Already Know C," but I'll try to be general enough for everyone. C++ was born at AT&T in the 1980s. It was a set of object-oriented extensions to C, an already popular language. The change from C's largely procedural view to C++'s object view marks a fundamental paradigm shift-one that changes how all programs and all programming problems are viewed.

Listing 1 demonstrates this sweeping claim. As you can see, this is a simple and perfectly correct portion of a C program. But, what is wrong with it?

The code is typical of C which publishes DATA L I B as a public structure. The logic that manipulates its members is sprinkled throughout many different application programs. If the DATA L I B structure was changed, every program using it would need to be altered or at least recompiled. With this procedural framework, knowledge is said to be *distributed.*

With C++, programs do not know or have direct access to members of a data structure. Instead, they call a function, specifically known as a *member function* or *method,* to retrieve members of the data structure.

Listing 1—A typical C program relies on distributed knowledge about data structures.

```
DATALIB DataLib;

while (getData(&DataLib)){
 printf("\nData Received at: %d:%d %d - %s",
         DataLib.Hour, DataLib.Minute, DataLib.Pressure,
         (DataLib.Pressure > DataLib.PrevPressure)
         ? "RISING" : "FALLING");
```

Although this represents cost in the number of instructions generated to achieve data-structure independence, it limits the dependencies to a few well-defined interfaces. The interfaces provide access functions to some of the data in the private section.

It isn't generally true that performance degrades overall by the object model. In some cases, the model allows for code generation that increases a program's performance. I'll sprinkle advice about the type of code C++ generates throughout this article.

Note: Data independence is not limited to C++. The same effect can be created using C or assembly language. A text file, encapsc.txt (available on the Circuit Cellar BBS), describes how to achieve the same effect in C.

Although I'll talk about how the switch to C++ represents a shift in thinking, I cannot provide a thorough, profound, and well-developed tutorial of a language as complex as C++ within the confines of a single article. At best, I can provide sufficient examples of the salient points of C++.

I'll begin with a practical example emulating an answering machine's behavior. Because it is a system with controls, inputs, and outputs, it offers similar types of problems to those found in most embedded applications.

However, let's start with the basics.

AN INTRODUCTION TO CLASSES

In C, a data structure would be defined and used as:

```
struct Date {
  int    month;
  int    day;
  int    year;
};

struct Date aDate;

aDate.month = 7;
aDate.day = 20;
aDate.year = 1969;
```

Just to review some basic C, memory is allocated for a structure called aDate, which is of type Date.

In C++, a programmer declares a class, which has a similar appearance (and to some extent, a similar function) to a data structure. A class declares both data and the functions that can access this class. These function members are known technically as *methods*. Listing 2 shows how a class is defined. Note that comments in C++ begin with two slashes and end with a carriage return.

The class definition shown in Listing 2 contains *public* and *private* sections. Anything listed publicly is accessible from anywhere or any program. The functions and variables from a private section can only be accessed from functions defined in the Date class.

In this example, the variables month, day, and year are private and can only be accessed by the functions declared in Date class. The functions Display and SetDate are public and may be called from anywhere. They control access to objects in the class.

The functions Date and ~Date are known as *constructors* and *destructors*, respectively. They are called automatically when an instance of the class is created or destroyed. These functions serve an invaluable purpose. Because of the constructors, data in a class can be initialized when created

Listing 2—C++ encapsulates data structure and behavior as shown in this Date class.

```
class Date {
public:
 Date(int m, int d, int y);              // constructor
 void Display();                         // display function
 bool SetDate(int m, int d, int y);      // set date
 ~Date();                                // destructor

private:
 int    month;
 int    day;
 int    year;
 char   holiday[30];
};
```

Listing 3—Here are examples of how (and how not) to use the Date class.

```
void main()
{
 Date startDate(7, 20, 1969);            // declare a Date
 Date endDate(99, 999, 9999);            // an invalid Date

 startDate.month = 6;                    // this is illegal
 startDate.SetDate(7, 20, 1994);         // set a date

 startDate.Display();
 endDate.Display();
}
```

and allocated resources can be freed when destroyed.

Listing 3 demonstrates how a program uses a class. Two Date objects are instantiated (created): startDate and endDate. Each declaration causes the constructor, the Date function, for this class to be called. The constructor initializes the object. Unlike other functions, constructors and destructors cannot fail and cannot readily report errors even if the parameters passed are wrong! Contructors have no way of returning errors. Because of this, it is imperative that constructors always initialize an object to a safe state, even when illegal parameters are passed.

The statement startDate.month is illegal because month is a private member of the Date class and cannot be directly accessed. One solution is to add a SetMonth method. As defined so far, a date can be set or displayed by using its public functions SetDate and Display.

Listing 4—Constructors initialize data but cannot explicitly return errors

```
// Constructor
Date::Date(int m, int d, int y){
 if (m < 1 || m > 12)          // if date is illegal
   m = -1:                     // indicate by a -1 in month

 month = m;
 day = d;
 year = y;
```

CONSTRUCTORS AND DESTRUCTORS

Instantly, a C programmer can recognize the value constructors and destructors provide. With them, an object always has the opportunity to properly initialize prior to its use. This, as with other C++ features, is far more important when an object is complex, containing linked lists and substructures. Constructors and destructors are part of the object model and are enforced by the language itself.

A typical constructor appears in Listing 4. The syntax Date::Date identifies this as a function belonging to the Date class. The class name appears to the left and is separated from the function or method name by double colons. Constructors always have the same name as the class to which they belong.

There can be, in fact, several constructors defined, each supporting different arguments types. This is a feature of C++ functions and methods and is not limited to just constructors. C++ matches function calls based on the argument list and not just on the function name. This way different member functions can be defined with the same name, but have different arguments. Listing 5 offers an example of this capability.

It is also possible to avoid having to declare functions for every permutation of calling parameters because C++ supports default parameter values as part of the calling convention. A default parameter value can be defined for any argument. When the argument is missing from a call, the default value is automatically inserted.

In Listing 6, the string argument in the Date constructor is defined to take on a default value of null. If the string argument is not passed during a call, a null value (the default value declared in the constructor's defini-

Listing 5-A class may have many constructors, depending on the arguments passed

```
class Date {
public:
  Date();
  Date(int m, int d, int y);
  Date(int m, int d, int y, const char *n);

private:
  int   month;
  int   day:
  int   year:
  char  holiday[30];
};

// constructor with no arguments
Date::Date0 {
  month = day = 1;
  year = 1994:


// constructor with mmlddlyy arguments
Date::Date(int m, int d, int y){
  month = m;
  day = d;
  year = y;


// constructor with holiday text argument
Date::Date(int m, int d, int y, const char *n){
  Date(m, d, y);
  strcpy(holiday, n);


void main0

  Date aDate;
  Date bDate(7, 20, 1994);
  Date cDate(1, 1, 1994, "New Year's Day");
```

tion) is supplied during the call. Default parameters are not limited to constructors.

Finally, you almost always need to create this next special case of a constructor for all of your objects. It would be highly desirable to create a new object by passing it a reference to an already existing object. For example, it is desirable to be able to initialize a date object with the value of another date object.

This type of constructor is called a *copy constructor* because the result is that the new object becomes a copy of the referenced object [see Listing 7).

Constructors are optional. If no constructor is defined, a dummy constructor is automatically created by the compiler. The dummy constructor is called but does nothing, not even initialize the data structure's contents. This dummy constructor's function is necessary for several reasons. However, it is mostly important for maintaining consistency in calling conventions when calling C++ functions from C or assembly language.

A destructor is called when a specific instance of a class is no longer within scope (i.e., when it will no longer be necessary, which is typically when a function terminates). Destructors are also optional and a dummy constructor is created by the compiler when it is not declared. A destructor has the same name as the class to which it belongs and is preceded by the ~ symbol, as in -Date.

Listing 6—*Optional arguments may be omitted on any C++ function.*

```
Date::Date(int m, int d, int y, const char *n = NULL) {
 month = m;
 day = d:
 year = y;

 if (n)
   strcpy(holiday, n);
}

void main0
{
 Date bDate(7, 20, 1994);      // NULL will be added
 Date cDate(1, 1, 1994, "New Year's Day");
}
```

CREATING CLASSES DYNAMICALLY

As with any C program, when an object is declared inside the scope of braces, allocation for it is typically made on the stack. The life of the object is only within the execution of the code in the braced section. Objects can also be instantiated within a program's global section or declared dynamically.

In C, dynamic allocation is managed through use of the ma l l o c and free functions. Space is allocated from the heap. These functions still work in C++, but they will not call the corresponding constructor and destructor. Instead, objects in C++ can be dynamically allocated using two new operators-new and del ete. Listing 8 demonstrates how these operators force the constructor and destructor to be called.

The new operator returns a reference to an object after allocating memory and calling the object's constructor. A null pointer is returned if the memory cannot be allocated. Because a pointer is returned, it must be used as a pointer. In C++, just as in C, members of a data structure are accessed by the -> notation when referenced by a pointer. The de **1 et e** operator calls the object's destructor before it frees the memory to the free store.

Listing 7—*Every C++* class *should also contain a Copy constructor.*

```
class Date {
public:
  Date(int m, int d, int y, const char *n);
  Date(const Date &someDate);
};

// copy constructor
Date::Date(const Date &someDate){
 month = someDate.month;
 day   = someDate.day;
 year  = someDate.year;
}

void main0
{
 Date aDate(7, 20, 1969):
 Date bDate(&aDate);
}
```

Listing &new *and de l e t e operators execute the constructors, but ma 1 1 o c doesn't.*

```
Date *mDate;
Date *pDate;

mDate = (Date *)malloc(sizeof(Date));  // no constructor call

pDate = new Date(7, 20, 1994);         // constructor call
pDate-> Display();

delete pDate;
```

Listing 9—*new* and *delete* can be used with *array definitions*

```
void main()

  Date anArray[20];          // constructor called 20 times
  Date * ap;

 ap = new Date[10];          // constructor called 10 times
 ap[5].SetDate(1, 2, 94);    // item 5 referenced

 delete [] ap;               // deletes entire array
```

The new operator is not limited to allocating classes or objects. It can allocate any defined type such as

```
int * pint;
pint = new int;
delete pint;
```

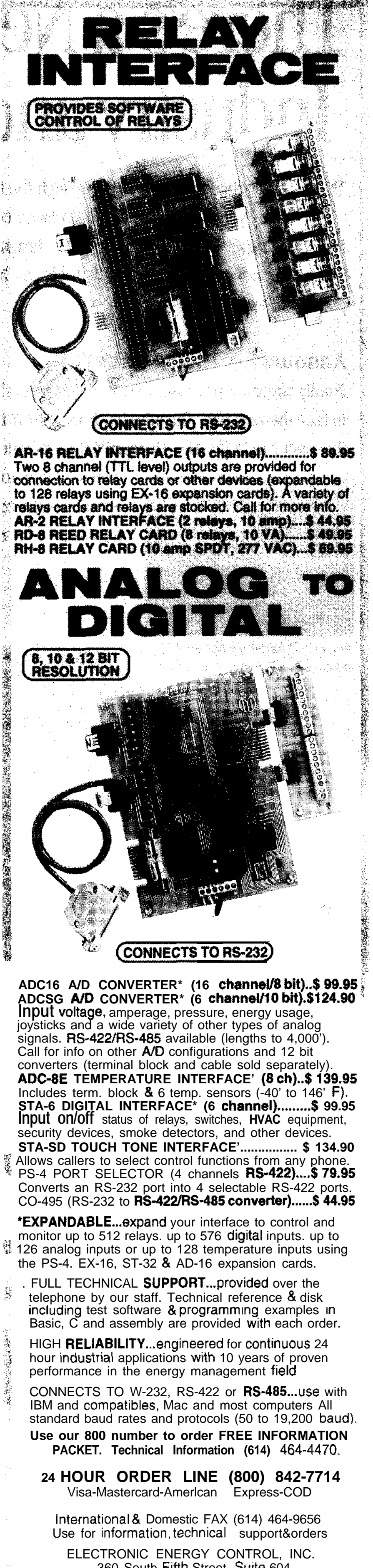
As you would expect, objects created with new and delete operators are persistent. They are not automatically deleted at the end of a function or even at the end of a program. (As a tangent, the behavior at the end of a program depends on the behavior of the operating system. In DOS and UNIX, conventional memory allocated by a program is automatically freed when the program terminates. In Windows, global heap memory remains.)

As Listing 9 illustrates, it is possible to create an array of objects. The constructor (and eventually the destructor) is called once for each element in the array of object definitions regardless of whether an object was created by a declaration or by the new operator.

Notice that to free the entire array you must use the symbol [] in the delete statement. On the surface, it might seem logical to presume that

Listing 10—*C++* simplifies this type *of C* program. Special *cases are handled* by subclassing

```
struct Salaried {
 float     salary;
};

struct Hourly {
 float     rate;
 float     hours;
};

struct Employee {
 int    paytype;
 char   employeeName[30];

 union {
   Hourly    hourly_pay;
   Salaried  salaried_pay;
 } u;
};

float ComputePay(struct Employee *emp)

 switch (emp->paytype){
 case HOURLY: {
   Hourly *p = &(emp->u.hourly_pay);
   return p->rate * p->hours;
   }
 case  SALARY:
   return emp->u.salaried_pay.salary;
```



#106

Listing 11—The C code in Listing 10 collapses into this much simplified C++ program.

```
class Employee {
public:
  .
  DisplayName();

private:
  char  employeeName[30];
};

class Hourly:public Employee {
public:
  . .
  float   ComputePay();

private:
  float  rate;
  float  hours;
};

class Salaried:public Employee {
public:
  .
  float   ComputePay();

private:
  float   salary;
};
```

the C++ would know that an array was declared and would therefore automatically remove the array. However, the language designers felt that there would be confusion over whether a program was referencing the lead object of an array or the entire array. The [] syntax specifically states that the entire array can be freed.

INHERITANCE AND POLYMORPHISM

Inheritance and *polymorphism* are areas where the power and elegant beauty of C++ hold substantial advantage. Used effectively, they can reduce a program's complexity, and with it, the size of the code generated. Inheritance is used to define an object's behavior as a superset of another object. Polymorphism takes advantage of method naming to make dissimilar objects behave logically alike. One cannot fully appreciate the effect of polymorphism without an example.

In C, the union construct identifies differing types of data that might be carried within a data structure. However, again, this is an example of where knowledge about how to handle this data structure is distributed. Each function must test for data types. Adding a new type becomes a time-consuming task of locating all cases where the code is affected. It is not uncommon to find this type of code in C (see Listing 10).

Instead of using unions and adding new data types, you should create different objects. New salaried types are supported by adding new object definitions. See Listing 11 for how the above listing would appear rewritten in C++.

The classes Salaried and Hourly both inherit the definitions of the Employee class. That inheritance is established by the syntax class Hourly:public Employee. Notice that each pay-type class has defined its own compute-pay method. That makes this code possible:

```
Salaried * s = new Salaried
  (...);
Hourly * h = new Hourly
  (...);

s-> ComputePay();
h-> ComputePay();
```

This example is not as powerful as the example which follows. However, it should be sufficient to convince you of the potential of compartmentalization. By relegating the code to specific objects, there is no longer a necessity for special-case code. Here is a more powerful example of the same code:

```
{
  int       k;
  Employee  *Ptr[20];
  Salaried  sEmp("Al Jones");
  Hourly    hEmp("John Doe");

  Ptr[0] = &sEmp;
  Ptr[1] = &hEmp;

  for (k = 0; k < max; ++k)
    Ptr-> ComputePay();
}
```

You can use a pointer to an Employee to point to a Salaried or Hourly employee. You can pass these pointers to functions and/or save them in data structures and arrays. Because they are pointers, they may be created dynamically. Once you have a pointer, you no longer care about its type as long as they share a common subset of method references.

The ComputePay methods would appear as:

```
float   Salaried::ComputePay()
{
  return  salary;
}

float   Hourly::ComputePay()
{
  return  rate  *  hours;
}
```

The current object reference is passed to ComputePay. This reference, known as the this argument, is taken from the object reference on call and is useful in some instances. For example, a method could return the current object reference by using the pointer:

```
Employee  &
  Employee::SomeFunction()
{
  return  *this;
}
```

OPERATOR OVERLOADING

Operator overloading permits the C++ compiler to change the behavior of most operators to fit the semantics of the objects on which they operate. For example, we presume that the addition operator works on integers and real numbers. However, we could define a `Fractions` class that would behave as follows:

```
Fractions a(1, 2);
Fractions b(1,4);
Fractions c;

c = a + b; // answer: 3/4
```

I won't go into greater detail on operator overloading here. However, I have posted samples of operator overloading in the BBS files. Because C++ permits overloading, it can redirect output as is shown in the next section.

Listing 12—*Operator overload redirects stream input or output.*

```
ostream& operator<<(const char *);
ostream& operator<<(const unsigned char *)
ostream& operator<<(const signed char *);
ostream& operator<<(char);
...
ostream& operator<<(short);
ostream& operator<<(unsigned short):
ostream& operator<<(int);
ostream& operator<<(unsigned int);
ostream& operator<<(long);
ostream& operator<<(unsigned long);
ostream& operator<<(float);
ostream& operator<<(double);
...

ostream& ostream::operator<<(double g){
 static char ascii[32];
 gcvt(g, 15, ascii);
 return ascii;


ostream& ostream::operator<<(signed char c){
 return operator<<((unsigned char) c);
```

cout AND cin

`cout` and `cin` are standard stream controls for C++. `cout` and `cin` behave much like `printf` and `scanf` do in C. You could use it by:

```
cout << "Hello," << 2 <<
     "the World! "
```

It prints "Hello, 2 the World!" on the stream device, which is typically the monitor. `cout` is used prevalently in C++, although `printf` and `fprintf` functions would work as do all of the other C function library functions. The advantage is that it is no longer necessary to embed `%s` and %d in the output statement. Someday, `printf` will appear as arcane as punched cards.

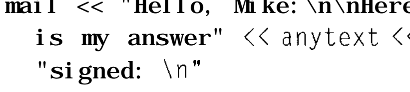
`cout` is defined as an object of class `ostream`, defined in `ostream.h` in your favorite compiler. To support this type of, functionality, the << operator must be overloaded for each acceptable data type. The output stream code eventually calls some code that converts the received data type to ASCII (Listing 12).

Arcane and off the point as all of this might seem, consider the following. Presume that an object is defined of type **Log**. It should be possible to use overloading to redirect output to this object:

```
Log logfile("abc");

logfile << "Hello," << 2 <<
  "the World!\n"
```

Although device redirection already exists, a **Log** object can be used to record a great deal more state information about your program. Finally, consider the same effect with a `Mail` object:

```
Mail mail("username", "1-508-
  555-1234");
```



```
mail << "Hello, Mike:\n\nHere
  is my answer" << anytext <<
  "signed: \n"
```

Having established some of the basics, we need to move on to a more real-world example.

A (MORE) REAL-WORLD EXAMPLE

This example is not of a real answering machine, but is a contrived example demonstrating design principles. Although everyone knows the basic operation of a telephone answering machine, converting that knowledge into C++ can prove to be a challenge for beginners. Like learning to drive a car, it's different when you have to navigate traffic.

A prototype diagram of an answering machine is shown in Figure 1. In addition to the announcement and recording tapes, the system consists of a volume slider and the buttons: On/Off, Play/Pause, Save, Erase, Record, and forward/reverse arrows. A message display shows the number of messages received.

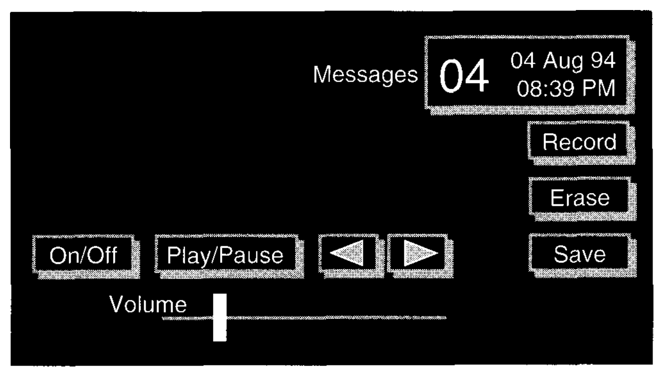


Figure 1—*Code on the BBS describes the behavior of this answering machine.*

Listing 13— *lastAction is an object reference and can be used to call member functions.*

```
Button  * lastAction;
. .

lastAction = &record;
.

if (playPause.IsButtonDown()){
 lastAction->Rewind();   // either tape
 lastAction->Play();     // either tape
```

Although this is a hardware-independent solution, it is only because no hardware has been developed-mind you, the 8051 would make an excellent chip to solve this problem. So, we'll just assume that we can make a function call to either C or assembler that will execute requirements such as enable recording.

The code for the system consists of one main loop waiting for something to happen such as the phone ringing or the Play/Pause button being pressed. An object is defined for a generic Button. The purpose of this object is to perform hardware-dependent functions such as reading the current button state.

Button is super classed by two more refined buttons: HoldButton and ToggleButton. The presumption is that the physical button has only an up or down state. ToggleButton treats the button as if it toggles back to the up position after its value is read. It does this by ignoring its physical state if it hasn't changed since the last read.

To read the value on an object such as a button, you could ask for its value. However, it may be smarter and more removed from the physical environment to ask whether the button is up or down:

```
if (playPause.IsButtonDown())
  {...}
```

The product's behavior demands rewinding and replaying either the greeting or recording tapes depending on which buttons are pressed. This is handled by maintaining the lastAction object reference. Listing 13 demonstrates how this is handled.

Well-crafted C++ programs give you a much better sense of the coding style and simplistic beauty of the design. I hope this introduction to C++ helps you understand some basic C++ principles that will eventually motivate you to learn the language.

Finally, both Borland and Microsoft have excellent development systems with integrated environments that you can play with. But regardless of what software package you have, remember there is no better and quicker way to learn than to just start coding.

Mike Podanoffsky has worked for over 20 years in computers, specializing in personal computers and database systems. He is currently working at Lotus Development on major database products. He is author of Dissecting DOS, *published by Addison-Wesley. He may be reached at mikep@world.std.com.*

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

401 Very Useful
402 Moderately Useful
403 Not Useful

Tight Budget '51 Emulation

8051 Family Emulator is truly Low Cost!

The DryICE Plus is a modular emulator designed to get maximum flexibility and functionality for your hard earned dollar. The common base unit supports numerous 8051 family processor pods that are low in price. Features include: Execute to breakpoint, Line-by-Line Assembler, Disassembler, SFR access, Fill, Set and Dump Internal or External RAM and Code, Dump Registers, and more. The DryICE Plus base unit is priced at a meager $299, and most pods run only an additional $149. Pods are available to support the 8031/2, 8751/2, 80C154, 80C451, 80C535, 80C537, 80C550, 80C552/62, 80C652, 80C851, 80C320 and more. Interface through your serial port and a comm program. Call for a brochure or use INTERNET. We're at info@hte.com or ftp at ftp.hte.com

Tighter budget? How about $149 emulation?

Our $149 DryICE model is what you're looking for. Not an evaluation board - much more powerful. Same features as the DryICEPlus, but limited to just the 8031/32 processor.

You can afford it!

So, if you're still doing the UV Waltz (Burn-2-3, Erase-2-3). or debugging through the limited window ROM emulators give, *call us now* for relief! Our customers say our products are still the best Performance/Price emulators available!

Look into our Single Board Computer solutions, too!

HiTech Equipment Corp
9400 Activity Road
San Diego, CA 92126
[Fax: (619) 530-1458]

Since 1983

(619) 566-1892

Internet e-mail: info@hte.com
Internet ftp: ftp.hte.com

#108

FEATURE ARTICLE

Rick Naro

Characterizing Processor Performance

A conspicuous lack of useful vendor information about optimizing system performance drives Rick to characterize microprocessor performance himself—a strategy he sees as critical to design success.

Microprocessor vendors often provide a great deal of documentation for their products. There are data sheets, user manuals, application notes.. . . Conspicuously missing, however, is useful information on optimizing processor and system performance.

Even if you design your embedded system hardware to run flat out and optimized performance is not a problem, there are still plenty of software design issues to consider. And, if you need to minimize the cost of a design-who doesn't in a high-volume embedded application—understanding the relationship between the CPU bandwidth, memory, peripherals, and software development tools is key to success.

The choices of cutting performance to achieve a lower design cost are many. You might vary the size of the microprocessor external bus paths to eliminate devices. You could add wait states and use slower memory devices which cost less than higher-speed devices. You could run the system clock at a nonstandard frequency, perhaps saving the need for an additional crystal oscillator. In applications calling for the high precision and dynamic range of floating-point arithmetic, you could use software emulation in place of an external math coprocessor.

Software design issues also affect the performance of the system. Choice of language, compiler, and memory model have a direct impact and must certainly be considered. An even more important consideration is how well the software is designed. If it doesn't use the most efficient algorithms and data structures, it could prove to be one of those applications that brings even the fastest computer to its knees.

To counteract the dearth of relevant documentation, this article offers a detailed look at the performance tradeoffs of the Intel '186 microprocessor family. Specifically, we'll be looking at the Intel '186EB and '188EB, which are used for all the timing measurements. While most of you likely use a different microprocessor family, many of the performance issues cross architectural boundaries. With a little imagination, you can apply these findings to your own design circumstances.

BUS BANDWIDTH

The Intel '186-family consists of 16-bit microprocessors with 16-bit internal data paths. However, when the first family members were introduced, Intel prepared two versions—the 8086 and the 8088.

For those who remember back to 198 1 when IBM was designing the first PC, you may recall that IBM made a conscious choice to use the 8088. Its use of an 8-bit external bus reduced hardware costs. Little has changed since then. You still have a choice of '186 and '188 family members where the only difference is the use of 16- or 8-bit external data paths.

As in 1981, a system designed around the '188 is less expensive since

Processor	Execution Time	Relative Performance
'188	28.105	0.639
'186	17.722	1.000

Table I--Even *though the 8-bit bus of the '188 is only half as wide as the 16-bit bus on fhe '186, the former achieves better than 60% of the /after's performance. Both systems are identical in all other aspects. All times are in milliseconds.*

EPROM Wait States	RAM Wait States	Execution Time	Relative Performance
0	0	17.965	1.000
0	1	1 8.748	0.958
1	0	20.157	0.891
1	1	20.968	0.857

Table **2—*The*** high *ratio of instruction fetches* **to data operations** *in a* ***16-bit*** *system shows* ***the*** *EPROM address space is more sensitive* ***to*** *waif* ***states*** *than* ***the*** RAM address *space. Tests were performed on a* ***16-MHz 80C186EB*** *and* ***all*** times are *in milliseconds.*

only half the number of memory devices (EPROM and RAM) are required. Further savings are gained by eliminating the extra data bus buffer.

If things were simple, we might expect the '188 to be exactly one half the speed of the '186 because of having half the bus bandwidth. But even back then, Intel built parallel CPU and bus interface units into the devices, complicating analysis. By running some test code on both processors, we can roughly determine the penalty of designing with an 8-bit external data bus (see Table 1).

This result shows that the 8-bit external version has nearly two thirds of the performance of the 16-bit version, which is considerably more than my initial speculation. Besides the separate CPU and bus units, both the '186 and '188 use an instruction queue to prefetch instructions-six bytes for the '186 and four bytes for the '188. On the surface, this seems to bias the results toward the '186 because of more instruction queue hits.

The solution to the problem lies in the bus bandwidth used rather than the available bandwidth. In the case of the '186, the bus interface unit is sitting idle more than 30% of the time while the '188 is chugging away at over 90% bus utilization. So, while the '186 bus unit is sitting idle, the '188 is busy catching up. Add more bus usage through DRAM refresh cycles, DMA cycles, and external bus masters, and the 16-bit external bus looks like a much better solution for higher-end systems.

MEMORY WAIT STATES

Wait states are used to match a fast processor with a slower memory or peripheral device. Normally, you want to run with zero wait states since this maximizes the system performance. But, in systems where excess bandwidth is available, designing in slower devices and inserting wait states is an acceptable compromise to reduce the cost of the system.

Since there are at least two distinct address spaces, the question of where to insert the wait states comes up. We can use the EPROM address space for code and constant data and the RAM address space for read/write data and the stack. To determine which option is better, we need to know the impact on throughput by inserting wait states separately into each address space and measuring the result. Memory devices can then be chosen to deliver a specific level of performance while reducing the memory device cost.

From Table 2, it is clear that the penalty for adding wait states to the RAM address space is only half that for EPROM address space. These findings make sense since the processor is constantly executing instructions, but not every instruction makes a reference to the data address space. With this knowledge, it becomes possible to optimize the wait states for each address space in the system with the cost and benefit known in advance.

DRAM REFRESH

We can also use dynamic RAM since we know the cost per bit is much less than for static RAM of the same density. For this scenario, we need to find the impact on performance of adding the additional refresh bus cycles to the normal mix and see what effect this has on the system.

While DRAM refresh has a low impact on the throughput, the lower cost of DRAMs must also be weighed against higher design costs associated with the additional hardware needed for RAS/CAS generation and timing.

Processor	Execution Time	Relative Performance
Refresh enabled	1 a.085	0.988
Refresh disabled	17.867	1.000

Table *3-Using* ***the 80C186EB, the*** *effect of DRAM refresh on a* ***16-bit*** *system* running *at* 16 ***MHz*** is *quite* ***small.*** *Whether or* **not** *fhe DRAM is used in a system is determined by* ***the*** *additional hardware cost of supporting lower-cost* ***DRAMs. All*** *times are in milliseconds.*

Some microprocessor vendors have recognized this and have optimized the external bus for a direct DRAM connection (e.g., NEC V35).

From my experience with bus utilization, I expect a 16-bit bus to be affected less than an 8-bit bus. A 16-bit bus has more idle bus bandwidth that can be handed over to the refresh controller without any impact on performance.

Still, there will be some impact since DRAM-refresh bus cycles have priority over other bus cycles and we need to perform refresh on a frequent basis. For instance, typically, there are 256 refresh cycles every 4 ms.

As you can see in Table 3, the penalty for DRAM is not bad, but there is one caveat to consider. The refresh overhead is fixed by the DRAM memory devices and is independent of the microprocessor. As you

Operation	Emulation	80C187	Relative Performance
Add float	226	17	13x
Add double	241	23	10x
Multiply float	275	17	16x
Multiply double	292	23	13x
Divide float	287	21	14x
sin(float)	306	27	11×
sqrt(float)	977	68	14×
Divide double	518	41	13×

Table *4-Here are common floating-point operations executed using an* ***80C187*** *math* ***coprocesser*** *and Borland's* ***C++*** *4.5 mafh coprocessor emulation. While hardware wins hands down, applications performing limited* ***floating-point*** *arithmetic can be we//-served by fhe* ***emulated*** *variety. All times in microseconds.*

slow the processor down or reduce the available bus bandwidth, the same number of refresh cycles must be performed in the same refresh interval. So, if YOU cut your bus bandwidth, expect to see the overhead of DRAM refresh increase.

Memory Model	Code Address Space	Data Address Space
Small	64 KB	64 KB
Medium	1 MB	64 KB
Compact	64 KB	1 MB
Large	1 MB	1 MB

Table *B-These are the most common* ***16-bit*** *real-mode* **memorv** *models encountered in a real-mode* ***IBM*** *PC or compatible. Tiny and huge memory models are left out as being unnecessary for the typical embedded system.*

FLOATING-POINT PERFORMANCE

Although the cost of floating-point hardware continues to drop, the decision to add a hardware math coprocessor is still an expensive proposition in any design. The alternative is software emulation of the math coprocessor. While this is more cost effective, it requires the availability of excess CPU throughput to take over from the missing hardware.

On the surface, the high floating-point penalty may appear insurmountable, but in the real world, an embedded controller doesn't spend anything close to 100% of its time on floating-point calculations. To decide if a software coprocessor can meet the system requirements, we need to know the difference in performance between the two implementations using the most common floating-point operations.

In addition to the comparisons between the floating-point operations in Table 4, it would help to know how much slack CPU is available. A system running near full capacity is not a candidate for a software emulation. However, sometimes spending money on a faster CPU and more memory to increase the available throughput to handle the software emulation can be the winning strategy that results in overall system-cost reduction.

COMPILER MEMORY MODEL

Enough on hardware! What about software design decisions that affect application performance?

Of course, the biggest contributors to efficient software are algorithms and design. Since these issues exist outside this article's scope, let's look at what we can control.

The Intel 80x86 microprocessors are famous (or infamous) for their use of segmented address space. Compilers, such as Borland C++, support a variety of memory models depending on the need to access 64 KB or 1 MB of the code and data address spaces. For those not familiar with the Intel architecture, four memory models are common. As you can see from Table 5, there are differences a design can exploit.

Recall in the section on memory wait states, we saw that the EPROM address space was more sensitive to wait states than the RAM address space. But, unlike wait states, the overhead for a 1 -MB code address space is only limited to the **CALL** and **RET** instructions using the longer segment and offset formats. However, predicting the behavior of the data address space is another matter and is hampered by complexity.

Memory Model	Execution Time	Relative Performance
Small	16.807	1.000
Medium	16.905	0.994
Compact	17.735	0.948
Large	17.869	0.941

Table **6**—*Comparing the relative performance of the same application in each memory mode/ on a* ***16-bit*** *system, the largest performance penalty comes from the use of far data pointers.* ***All*** *times are in milliseconds.*

Local variables allocated in registers or on the stack are always accessed without penalty as is most statically allocated data. The penalty arises when far pointers are used. Not only are more pushes and pops required to pass parameters, the actual accessing of the data also takes longer with the need to load a segment register. From Table 6, we can see that the results of running the same application in each memory model reflect this additional overhead.

The penalty for having a large code address space is insignificant. But, the large data address space costs 5% of the total bandwidth.

The moral of the story is to stick to the small or medium memory models. Use far pointers selectively when access to more than 64 KB of data is required.

UNDERSTANDING INTERRUPT LATENCY

Interrupt latency involves the delay in responding to an event and has several components-the time to complete the current instruction, the time to save the processor state on the stack, and the time to get to application code where the interrupt is finally serviced. The balance of the time spent servicing the interrupt is the interrupt service time.

Although the first two delays are out of our hands, the time it takes to get in and out of the interrupt service routine is ripe for optimization. It is important to know just what the interrupt latency of a high-level language is so you can decide if an assembly language routine improves performance.

Modern compilers like Borland C++ and Microsoft Visual C++ perform a great deal of optimization. But both compilers always push the entire processor state on the stack, even if only a fraction is actually used.

On the test '186EB system, a C++ interrupt handler with a single I/O command takes a total of 15.4 µs to execute. If the same code is rewritten in assembly language, the time can be reduced to just 7.7 µs. It is worth noting that the assembly language advantage is temporary. More complex interrupt handlers require you to save more of the processor state, which eventually equalizes the overhead.

Still, in my opinion, great assembly language programmers have an edge over the compiler in writing optimized code.

HEAP PERFORMANCE

Many embedded-system developers try not to think about heaps. They avoid them as much as possible due to their nondeterministic run-time requirements. While fixed-size allocation speeds up the time required to allocate and deallocate memory, the hottest trend toward object-oriented programming in embedded systems is likely to force programmers to consider the effects of heaps.

Unlike C, C++ includes dynamic memory allocation in the language specification, so it is difficult to avoid. While it is possible to create a C++ application using only statically allocated objects or objects created on the stack, knowing that the new and delete operators are available solves some thorny development issues. If you plan to use these functions, it is best to know in advance what the best, worst, and average times for heap-based objects is.

For a simple test, you can allocate and delete array objects from the heap in a random fashion, measuring the overhead over a period of time. Based on this information, the software design optimizes performance by preallocating time-sensitive memory and lets the noncritical code take advantage of the efficiency of dynamically allocated memory. To test system performance, 500 blocks of random size are randomly allocated and released.

Operation	Best Case	Worst Case	Average
new	66.9	718.4	93.0
delete	49.4	180.8	57.4

Table 7—*Heap-based dynamic memory allocation is not exact or predictable due to the use of linked lists. Shown here are the best, worst, and average execution times to allocate and free memory in a C++ application which randomly allocates and frees objects from the heap. All times are in microseconds.*

From the results in Table 7, it appears that freeing up dynamically allocated memory is more efficient. This is as it should be since the block size plays no role, unlike when the block is first allocated. While the new operator's average behavior is not far from the best-case behavior, its worst case sticks out like a sore thumb.

While one may not be able to avoid a heap, it certainly is possible to live with one. Third-party replacements for the Borland and Microsoft dynamic memory management packages are available with options for fixed-size allocation, multiple heaps, and the ability to catch heap errors. Even the C++ language recognizes that you might not be able to live with the default memory allocators, so they offer a custom memory allocator more suitable for a real-time system. As a last resort, you can simply avoid the use of the new and `delete` operators.

PUTTING IT ALL TOGETHER

I covered many of the most accessible hardware and software optimizations that impact system performance. Unfortunately, the specific data provided may be of little use unless you're one of those lucky souls designing with Intel and AMD '186-family processors or the NEC V-series microprocessors.

Nonetheless, the design optimization techniques are general purpose enough that they can benefit any hardware designer or software developer. To understand the key to cost-effective design, you must get to know your system components inside and out before making any design assumptions. That way, both wild and educated guesses can be transformed into sure bets. And, the resulting design will certainly be a success. ❑

Rick Naro is president of Paradigm Systems, a developer of embedded system development tools for the Intel/AMD 186 and NEC V-series microprocessors. In the past, he designed hardware and wrote applications for embedded 80x86 systems. He may be reached at 73047.3031@compuserve.com.

IRS

404 Very Useful
405 Moderately Useful
406 Not Useful

OUR SMALLEST EPROM EMULATOR

Eliminate the need to *burn and learn*
all in a package about the size of a 9 volt battery

- Super small (about 2"x 1 "xl ")
- Uses Surface Mount Design
- Fits in EPROM socket
- Downloads in less than 7 seconds
- Emulates 2764, 27128, 27256, 275 12
- Access time < 1 OONS
- Plug and Play: Plug cable into computer and download. Once power is supplied to circuit, cable can be removed
- Includes software for IBM compatable PC's
- Loads Intel Hex and Binary files
- Includes Serial Cable & Battery Backup
- Nearly same footprint as EPROM

411 Washington Street,
Otsego, Michigan 49076
TEL: 616-694-4920 FAX: 616-692-2651
Since 1985

CALL NOW for Your Special Introductory Price of $194.95

***plus** $5.00 shipping and handling*

VISA & MASTERCARD ACCEPTED

#109

Designing with PC/104

FEATURE ARTICLE

Rick Lehrbaum

As PC architecture takes over the embedded world, there's greater need for standarization. Why? Lower costs, longer product life, PC and modular compatibility, fewer development tools...

Over the past ten years, the IBM PC-compatible architecture has become an increasingly popular platform. In addition to their typical use as dedicated desktop computers, they've reached into the embedded world. They're now being used in embedded microcomputer applications such as vending machines, laboratory instruments, communications devices, and medical equipment. PCs are beginning to be found everywhere!

THE TREND TOWARD EMBEDDED PCS

From a computer architect's perspective, the PC architecture with its 808%based origins and inherently segmented world view is hardly something to get excited about.

Why, then, turn the world's favorite ***desktop*** system into an ***embedded*** microcomputer standard? Why not just keep using a Z80, 68HC11, or 8051?

Regardless of its particular implementation-from 4- and 8-bit single-chip microcontrollers to high-performance, 32-bit RISC processors—embedded microprocessors are simply a means to an end-not an end in themselves. After all, the purpose of an embedded microcomputer is to run the application software. It's the software, not the embedded computer, that makes the application what it is. As long as it can run the application software acceptably, the ideal embedded computer is one that minimizes risks, costs, and development time.

Development cost is the major reason for shying away from a multiplicity of microprocessors since their architectures vary greatly. Each requires new development tools, including emulators, compilers, and debuggers. And, every time you use a different microprocessor in a system design, you'll invest thousands of dollars and weeks of time putting the development environment in place. No wonder system developers seek alternatives to using the latest new microprocessor in every new project.

Also, it's common for old projects, based on older microprocessors, to

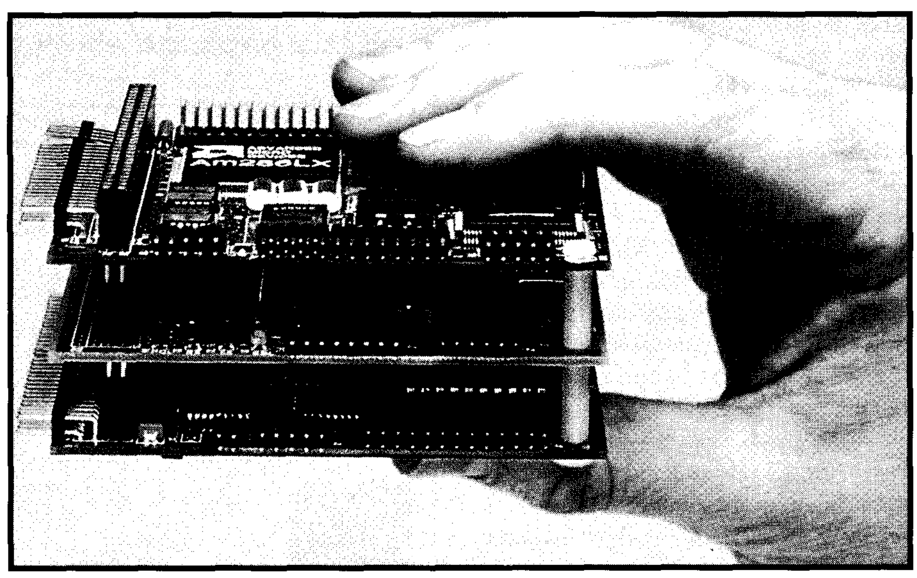


Photo 1—PC/104 modules are compact, rugged, and self-stacking. This three module stack measures just 3.6″ × 3.8″ × 2″, yet it contains fhe equivalent functions of a complete desktop PC: a PC/AT motherboard, up to 16 MB of system DRAM, serial and parallel interfaces, Ethernet LAN controller, SVGA display controller, and a bootable solid-state disk drive.

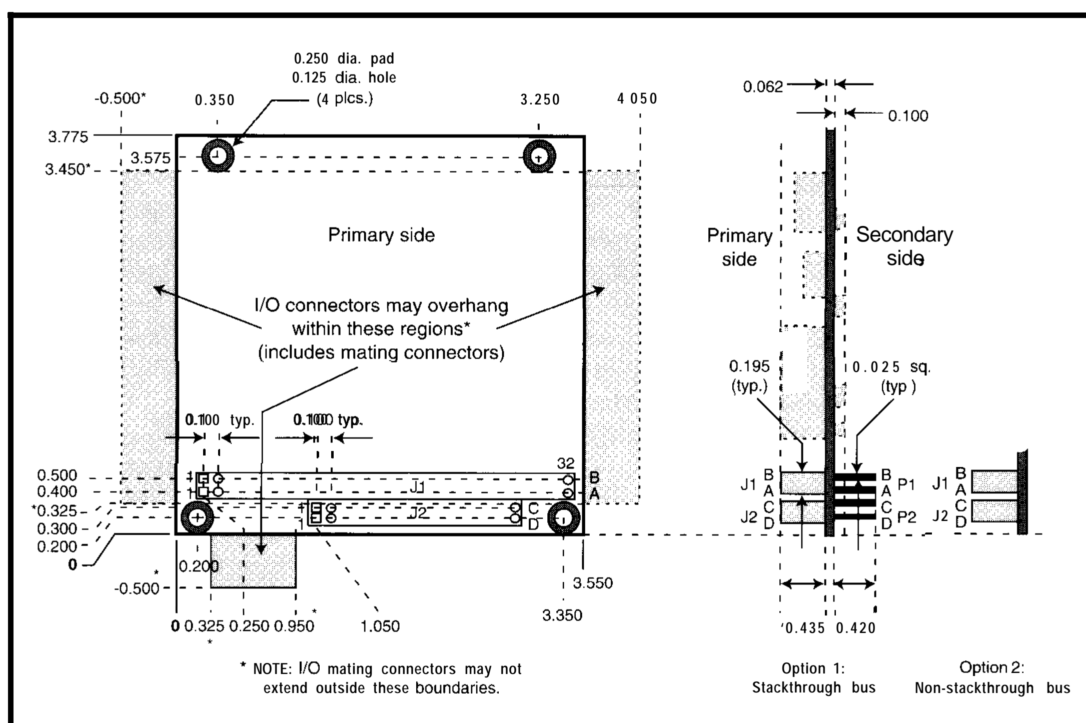


Figure 1—*A dimension drawing, extracted from the PC/104 specifications, shows the detailed mechanical dimensions of the 16-bit PC/104 module format. The P2/J2 connector is not required on 8-bit modules, but may be included as an option to provide "pass through" of a full 16-bit bus.*

become difficult or even impossible to maintain, as familiarity with the older architectures and their development tools fades.

All this has stimulated a desire for hardware and software standards. On the software side, this means using C, C++, and object-oriented programming methods. Programmers increasingly rely on familiar software environments such as UNIX, DOS, or Windows, and interface standards like TCP/IP, GUIs, and so on.

But what about hardware standards? Unfortunately, the tremendous diversity of microprocessor architectures, from the lowly 8051 to the high-end RISC CPUs, has prevented the emergence of any real standards for embedded-system hardware. Only the industrial computer buses such as VME, Multibus, and STD provide a measure of consistency. However, their use is limited to systems which are larger and more complex (and therefore less cost-sensitive) than most typical embedded systems.

On the other hand, the highly multisourced PC-compatible '386/'486 CPUs, chipsets, and associated peripherals have made the PC architecture attractive as a cost-effective hardware platform for low- and medium-performance applications.

With over 200 million desktop PCs in use worldwide and nearly a million new ones sold *each week,* the PC architecture has been dubbed the *Industry Standard Architecture* (ISA).

This is why the PC architecture is gaining increasing acceptance as an embedded microcomputer standard. Using an embedded-PC architecture leads to significant savings in development time and money. PC development tools are plentiful, cost-effective, and easy to use. PC-compatible chipsets and peripherals are abundant. Their functions are familiar and well documented. PC-oriented software components are readily available and include real-time OSs, drivers, function libraries, and application programs. Hardware engineers know the PC's bus and programmers, its software.

THE "ITSS PRINCIPLE"

In short, the reason so many embedded system developers are migrating to the PC architecture lies not in the hardware, but in the software. This trend has inspired the ITSS principle, a new "law" of embedded system engineering, which stands for *It's the software, stupid!*

MAKING THE PC FIT

One potential problem with using the PC architecture in an embedded system is that standard PC subsystems don't meet the more stringent size,

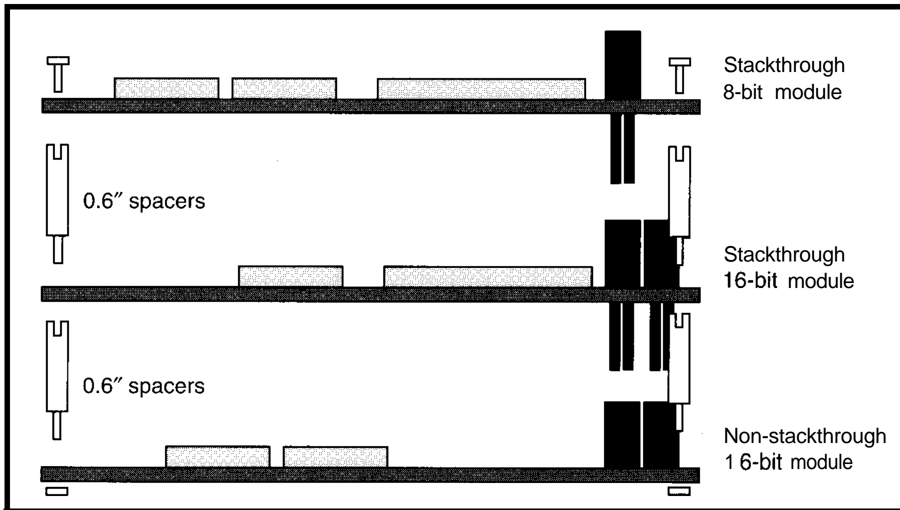


Figure 2—*Multiple PC/104 modules stack directly on top of each other using self-stacking pin-and-socket bus connectors. Four spacers rigidly attach each module to the one above and below it.*

power, ruggedness, and reliability requirements of most embedded applications. This is natural since PCs are optimized for the highly price-sensitive desktop personal-computing market.

But, you can avoid this problem by designing a custom, chip-level embedded PC directly into the embedded system's hardware. This way you can take advantage of PC chipsets, components, and software in an embedded environment.

The trouble with this approach is that it doesn't eliminate many of the costs and risks you want to avoid by using an off-the-shelf PC architecture. You still end up designing and debugging a CPU subsystem, licensing and porting a BIOS, and in many other ways needlessly reinventing the wheel.

Since standard PC subsystems aren't well-suited to the targeted environments, the desire to use PC architecture in embedded systems thus contains an inherent contradiction. This is what inspired the creation and rapid acceptance of the PC/104 embedded-PC modules standard (see Photo 1).

WHAT IS PC/104?

PC/104 offers full hardware and software compatibility with the standard desktop PC (and PC/AT) architecture, but in an ultra-compact (3.6" x 3.8"), self-stacking, modular form. PC/104 defines a standard way to repackage desktop PC functions for the ruggedness and reliability constraints of embedded systems. Consequently, PC/104 offers an attractive PC-compatible alternative to traditional microprocessor-based embedded systems.

Although PC/l 04 modules have been around since 1987 (in the form of Ampro's MiniModules), it was not until Ampro released a formal specification to the public domain in 1992 that interest in PC/104 skyrocketed. Since then, hundreds of PC/104 modules have been announced by the more than 140 members of the nonprofit PC/ 104 Consortium. In 1994, PC/104 achieved a significant milestone when Intel endorsed it as a recommended way to expand designs based on Intel's new embedded '386 CPUs.

In 1992, a working group of the IEEE embarked on a project to standardize a small form-factor version of the PC/AT bus, which was also based on PC/104. The new IEEE "P996.1" draft standard, which conforms closely to PC/104's specification, is now approaching IEEE approval.

WHAT'S IN THE PC/104 STANDARD?

As mentioned above, the key differences between PC/104 and the normal PC hardware standard are mainly mechanical. Instead of the usual PC or PC/AT expansion card form-factor (12.5" x 4.8"), each module's size is reduced to approximately 3.6" x 3.8".

Two bus formats for 8- and 16-bit modules are provided. However, unlike the 8- and 16-bit versions of the normal PC bus, 8- and 16-bit PC/104 modules are the same size. Figure 1 shows the detailed mechanical dimensions of the 16-bit PC/104 module format. An 8-bit module has no P2/J2 bus connector.

To eliminate the complexity, cost, and bulk of conventional motherboards, backplanes, and card cages, PC/104 modules implement a self-stacking (also referred to as *stackthrough)* bus connector. Multiple modules are stacked directly on top of each other without additional bussing or mounting components. Four nylon or metal spacers (0.6" in length] are normally used to rigidly attach the PC/104 modules to each other as shown in Figure 2.

Rugged and reliable 64- and 40-position male/female header connectors replace the standard PC's 62- and 36-position (P1 and P2) edge-card bus connectors. The PC/104 bus connectors feature two pin-and-socket rows on 0.1" centers and normally have gold-plated contacts. Both Samtec and Astron, two connector companies,

Pin Number	J1/P1 Row A	J1/P1 Row B	J2/P2 Row C[1]	J2/P2 Row D'
0	—	—	o v	o v
1	IOCHCHK*	o v	SBHE*	MEMCS16*
2	SD7	RESETDRV	LA23	IOCS16*
3	SD6	+5 v	LA22	IRQ10
4	SD5	IRQ9	LA21	IRQ11
5	SD4	-5 v	LA20	IRQ12
6	SD3	DRQ2	LA19	IRQ15
7	SD2	-12 v	LA18	IRQ14
8	SD1	ENDXFR*	LA17	DACKO*
9	SD0	+12 V	MEMR'	DRQO
10	IOCHRDY	(KEY)*	MEMW*	DACK5*
11	AEN	SMEMW	SD8	DRQ5
12	SA19	SMEMR*	SD9	DACK6*
13	SA18	IOW*	SD10	DRQ6
14	SA17	IOR*	SD1 1	DACK7*
15	SA16	DACK3*	SD12	DRQ7
16	SA15	DRQ3	SD13	+5 v
17	SA14	DACKI*	SD14	MASTER*
18	SA13	DRQ1	SD15	o v
19	SA12	REFRESH*	(KEY)*	o v
20	SA1 1	SYSCLK	—	
21	SA10	IRQ7	—	
22	SA9	IRQ6	—	—
23	SA8	IRQ5	—	
24	SA7	IRQ4	—	
25	SA6	IRQ3	—	—
26	SA5	DACK2*	—	—
27	SA4	TC	—	—
28	SA3	BALE		—
29	SA2	+5 v	—	—
30	SA1	o s c		—
31	SAO	o v		—
32	o v	o v	—	—

NOTES:
1. Rows C and D are not required on 8-bit modules, but may be included.
2. B10 and Cl9 are key locations.

Table 1—*The PC/104* names comes *from the use of* 104 *bus signals. Each* PC/104 *bus* signal is *equivalent* to *a corresponding signal of the normal PC/AT bus.*

currently offer alternate sourcing of the approved bus connectors.

PC/IO4 bus signals are functionally identical to their counterparts on the PC/AT bus. Their assignments to the 104 positions on the PC/104 header-bus are listed in Table 1.

To reduce power consumption to around I-2 W per module and minimize chip count, the bus drive was reduced from the normal PC's 24 mA to 4 mA. This permits HCT logic and many VLSI ICs to directly drive the bus without additional buffer chips.

Many developers wonder how many modules can be used on a single PC/104 bus. The answer is not simply related to PC/104's reduced bus drive current. Actually, the low 4-mA drive does not result in a small number of permissible bus modules. For most embedded systems, there is plenty of bus drive. In fact, since the maximum input load spec is 0.4 mA per bus signal, a 4-mA bus drive current can theoretically handle ten bus loads!

In practice, factors such as signal trace lengths and connector impedance transitions limit the number of modules you can reliably use to between six and eight. The actual limit, for a particular system, depends on total bus length, number of stacked connectors, environmental issues, and the specific modules used. Also don't forget to consider voltage drops on the bus power signals due to multiple stacked modules.

Bus termination is an option, as well. If you plan to terminate a PC/104 bus, be sure to use the AC method of termination defined in the PC/104 specification rather than pure resistive termination.

Plain resistive termination, usually 220/330 Ω between each signal and ground, exceeds available bus current. On the other hand, the recommended AC termination consists of a series R/C network between each signal and ground. This approach draws no static current and provides a better impedance match for the bus.

If you're not sure whether or not termination is needed in your system, it's best to provide a way to add it later. A number of PC/104 vendors offer special plug-in PC/104 terminators, which provide the method of AC termination recommended by the PC/104 spec. These terminators can be added at any PC/104 bus stacking location. You can also include positions for tiny SIP termination networks directly on PC/ 104 modules or interfacing boards you design.

When you use the PC architecture in embedded applications, it's not uncommon to run out of bus interrupt channels. This is especially true of byte-oriented (8-bit) interfaces such as serial ports because the 8-bit subset of the PC bus contains six interrupt lines, most of which are dedicated to standardized system functions. Unfortunately, since the bus interrupt lines are active high, the common technique of wire-ORing multiple interrupt requests on a single-interrupt input line (used with other buses) is not possible.

To circumvent this problem, the PC/104 spec includes a recommended means for multiple interrupting sources (on one or more modules) to share a single bus interrupt. A sample interrupt-sharing circuit appears in Figure 3.

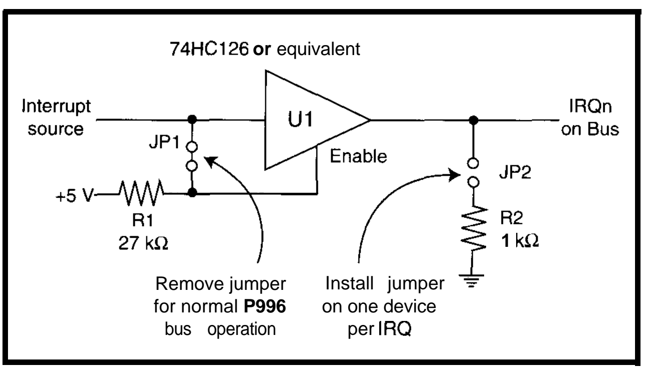


Figure 3—*This schematic shows a typical means of implementing the PC/104 bus interrupt-sharing option. While interrupt sharing is not required, if is frequently provided by 8-bit PC/104 modules that implement communications and nefworking functions.*

PC/104 IN REAL APPLICATIONS

Although configuration and application possibilities for PC/104 modules are practically limitless, there are a few ways the modules tend to be used in actual embedded systems.

- Stand-alone module stacks

As illustrated in Figure 4, stacks of PC/l 04 modules can be used like ultracompact bus boards, but without the usual requirement for backplanes or card cages. Often, a PC/104 module stack is bolted somewhere inside the embedded system's enclosure in a convenient location that would otherwise simply be dead space. In this manner, an entire PC can be embedded directly within a system that would otherwise require an external PC.

There are also a variety of off-the-shelf PC/ 104 stack enclosures that host from three to six PC/104 modules. Enclosed PC/104 stacks like these can be self-contained systems or can be used as subsystems within larger systems. These PC/104 system enclosures are designed for a variety of environments (commercial, industrial, and vehicular) and are available with options like PC/104 form-factor power supplies (for 8-80-V AC/DC inputs), shock mounts, and quick-release mechanisms.

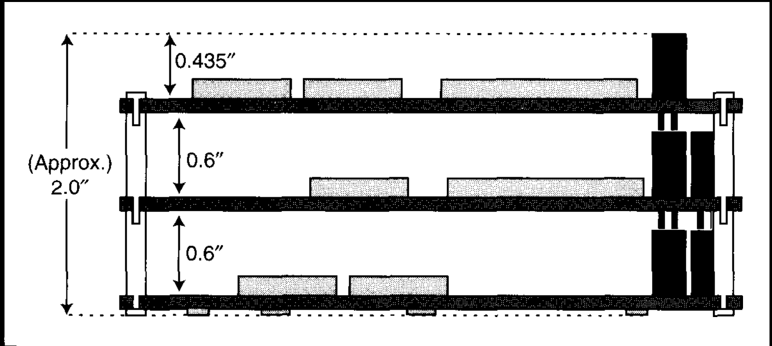


Figure 4—*PC/104 modules can be used as stand-alone stacks with all required system functions provided by PC/104 modules stacked together. In fhis approach, the modules function like a miniaturized backplane bus.*

• Macrocomponent applications

In Figure 5, another common method of using PC/104 is shown. Here it is used as macrocomponents that plug into a custom, application-specific baseboard. The PC/104 baseboard typically contains all interfaces and logic that aren't available (or desirable) on the PC/104 modules. Typically, the baseboard includes power supply components, signal conditioning, external I/O connectors, and so on.

What's interesting about the macrocomponent approach is instead of plugging the I/O into the computer, you plug the computer into the I/O! It's a new embedded-system paradigm. This approach lets you focus more energy on the application's unique requirements, and less on (re)inventing a basic (micro) computer architecture. With this approach, the system becomes a hybrid of out-sourced modules (the PC/104 modules) plus a custom-designed board (the baseboard).

Often, the baseboard provides multiple PC/104 stack locations. This means that the modules can be distributed horizontally, thereby keeping a low profile so there's room for upgrades and expansion in the future. Whenever possible, leave extra vertical space (at least 0.6") so the PC/104 module's self-stacking bus can be used for future upgrades and options. This space also provides room for temporary addition of modules for system debug, test, and service.

The shape and size of the baseboard is completely arbitrary. The baseboard typically takes the shape of the desired end system, so its shape can be anything-square, round, rectangular, customized.

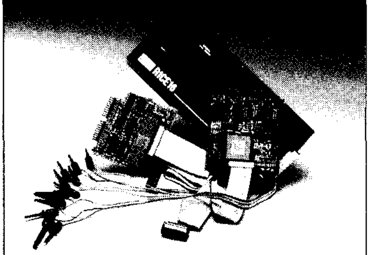
• Mezzanine bus applications

A third and increasingly common way for PC/104 modules to be used is as I/O expansion daughter modules on PC-compatible single-board computers (SBCs). This approach, known as a PC/104 *mezzanine bus,* is now found on nearly every new PC-compatible SBC, including both stand-alone (proprietary form-factor) SBCs and passive backplane (PC-expansion-card form-factor) industrial PCs.

PIC16C5x/16Cxx Real-time Emulators

Introducing RICE16 and RICExx-Juniors, real-time in-circuit emulators for the PIC16C5x and PIC16Cxx family microcontrollers: affordable, feature-filled development systems from **$599** *

*Suggested Retail for U.S. only



RICE16 Features:

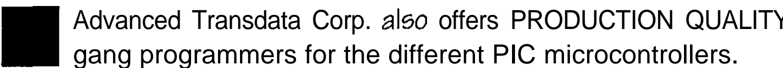
- Real-time Emulation to 20MHz for 16C5x and 10MHz for 16Cxx
- PC-Hosted via Parallel Port
- Support all oscillator type5
- 8K Program Memory
- 8K by 24-bit real-time Trace Buffer
- Source Level Debugging
- Unlimited Breakpoints
- External Trigger Break with either "AND/OR" with Breakpoints
- Trigger Outputs on any Address Range
- 12 External Logic Probes
- User-Selectable Internal Clock from 40 frequencies or External Clock
- Single Step, Multiple Step, To Cursor, Step over Call, Return to Caller, etc.
- On-line Assembler for patch instruction
- Easy-to-use windowed software

Emulators for **16C71/84/64** available now!

- Support 16C71, 16C84 and 16C64 with Optional Probe Cards
- Comes Complete with TASM16 Macro Assembler, Emulation Software, Power Adapter, Parallel Adapter Cable and User's Guide
- 30-day Money Back Guarantee
- Made in the U.S.A.

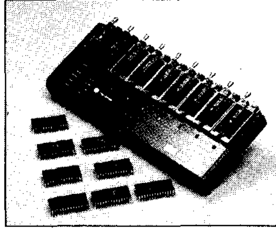
RICE-xx Junior series

RICE-xx "Junior" series emulators support; PIC16C5x family, PIC16C71, PIC16C84 or PIC16C64. They offer the same real-time features of RICE16 with the respective probe cards less real-time trace capture. Price starts at $599.

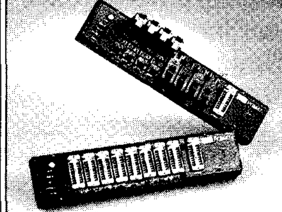


Advanced Transdata Corp. also offers PRODUCTION QUALITY gang programmers for the different PIC microcontrollers.

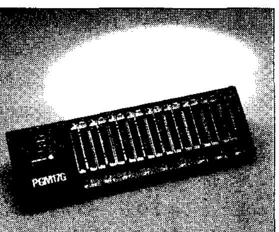
▪ Sand-alone COPY mode from a master device ▪ PC-hosted mode for single unit programming ▪ High throughput ▪ Checksum verification on master device ▪ Code protection ▪ Verify at 4.5V and 5.5V ▪ Each program cycle includes blank check, program and verify eight devices ▪ Price5 start at **$599**



PGM16G: for 16C5x family



PGM47: for 16C71/84



PGM17G: for 17C42

Call (214) 980-2960 today for our new catalog.

For RICE16.ZIP and other product demos, call our BBS at (214) 980-0067.

Advanced Transdata Corporation Tel **(214) 980-2960**
14330 Midway Road. Suite 128. Dallas, Texas 75244 Fax (214) 980-2937

#111

OBJECT-ORIENTED HARDWARE

Using PC/104 modules as macrocomponents parallels the object-oriented software methods of most of today's programmers. In object-oriented software, the program is broken into building blocks which are separately specified, developed, tested, and maintained. Object-oriented software greatly reduces the risks and complexity of software development and accelerates project schedules while producing more powerful, feature-rich, and maintainable application software.

Similar benefits are realized when PC/104 CPU and I/O macrocomponents are used as the building blocks of object-oriented hardware. And, you experience similar rewards-projects are completed faster, at lower budgets, with enhanced features, and are considerably easier to maintain.

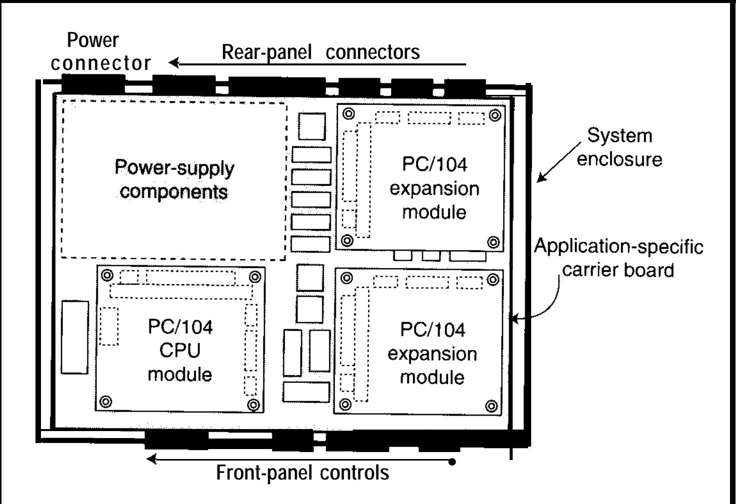


Figure 5—*Many successful users of PC/104 modules treat them like macrocomponents, plugged into an application baseboard. In this approach, the baseboard usually contains all functions that are unique to the specific application, and the PC/104 modules provide standard PC system functions such as CPU, mass storage, networking, communications, and display interface.*

MAKING THE MOST OF IT

There are some techniques of exploiting an object-oriented, module-based approach to embedded system design that can help you make the most of using PC/104 modules.

Make the PC architecture a macrocomponent. The entire embedded-PC architecture can be a single plug-in component, including all motherboard functions, system RAM, memory, and BIOS. You shouldn't need to be concerned with licensing or modifying a PC BIOS. Your PC/1 04 CPU module can include a solid-state disk, so you also don't have to worry about ROMing your embedded application's code.

Let variable performance work to your advantage. Projects frequently end up needing more CPU performance than originally anticipated. When this happens, be prepared to unplug the PC/ 104 CPU module

#112 See us at Embedded Systems East booth #510

you're using and replace it with a faster one. Can you imagine doing this with an 8051, 68HC11, or a discrete 80386SX?

Also, keep in mind the option of kick-starting a project by initially using a faster PC/104 CPU module than required to get the application up and running quickly. Later, you can cost reduce by optimizing the software and substituting a slower (and less expensive) CPU module.

To anticipate these possibilities, select PC/104 CPU modules that are members of a CPU family offering a broad range of CPU types and performances.

Allow for performance and feature options and upgrades. No doubt, you've been in the position of having to provide both high performance and low cost within one design. Now, you can provide both by offering multiple price and performance options. In a PC/104-based system, a single base design supports multiple feature or cost configurations. You can offer '486 performance at the high end and 8088 economy at the low end. You can provide a wide variety of communications options based on PC/104 serial, modem, Ethernet, or even wireless LAN plug-in modules.

Take advantage of sophisticated PC functions. In contrast to traditional microcontroller-based designs, your PC/104-based system design can draw on a rich set of PC technologies. Your designs need not be limited by what you can do yourself!

Here are some readily available options:

- user-friendly graphical user interfaces (GUIs) instead of character or LED displays
- popular mass storage devices (floppy, IDE hard disks, SCSI drives, or PCMCIA cards with flash-file-system support software) instead of ROM or battery-backed RAM
- full-function LANs (Ethernet, Arcnet, Token Ring) instead of slower RS-232 or RS-485 multidrop interfaces
- various SCSI or PCMCIA devices
- a wide range of off-the-shelf, application-oriented PC/104 modules (digital and analog I/O, motion control, etc.) complete with ready-to-use PC-compatible software drivers

Maximize system life expectancy. A system based on PC/104 modules has a longer life span than that of a traditional monolithic embedded system. Some of this longevity stems from the fact that when a monolithic system design no longer meets the requirements of its users, you may be faced with a redesign.

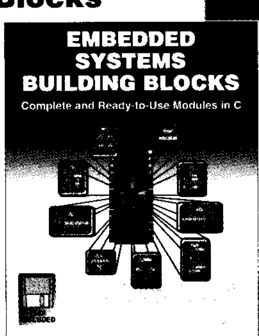
On the other hand, with a module-based system, it is often possible to upgrade the system to a higher performance CPU, faster or alternative peripheral interfaces, and enhanced software. Thus, a PC/104-based system design might survive two or three times longer than a monolithic one.

If you must replicate or service a particular design over several years, the risk of component obsolescence becomes an important issue. Monolithic designs are in trouble when one of the chips used in the system is no longer offered by its manufacturer.

All You Need for Building Embedded Systems

These two books, with the companion code disks, provide a complete operating system kernel and reusable program modules coded in C

Embedded Systems Building Blocks
by Jean J. Labrosse



Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C contains software modules you can use to design embedded systems and explains how to use these modules and modify them as needed. Labrosse provides highly portable, fully functioning code for many common processes: keyboard scanning, display interfaces, timers and clocks, discrete I/O, analog I/O, and serial communications. Labrosse provides basic building blocks for all these processes freeing you to work on the fun and unique parts of your designs.

R&D Publications, 1995, 620 pp. ISBN 0-13-359779-2
V74 with disk. $49.95

μC/OS
The Real-Time Kernel
Jean J. Labrosse

μC/OS *by Jean J. Labrosse*

This book explains the design and implementation of the Micro-Controller Operating System, a portable, ROMable, preemptive, real-time, multitasking kernel for microprocessors. The system is written in C with assembly language code for the target microprocessor kept to a minimum. It can be ported to any microprocessor that provides a stack pointer and allows the CPU registers to be pushed onto and popped from the stack. The system can manage up to 63 tasks, with performance comparable to many commercially available kernels. The text explains the fundamentals of multitasking real-time systems, details the design decisions of this kernel, and includes a user's manual for the system.

R&D Publications, 1992, 266 pp. ISBN 0-13-031352-l
W62 with disk. $54.90

R&D Technical Books Satisfaction Guaranteed

ORDER TODAY!

913-841-1631 FAX 913-841-2624
e-mail: rdorders@rdpub.com
All orders must be prepaid in US dollars by check, money order, or credit card MasterCard, VISA, and American Express are accepted.

FREE R&D Technical Book Catalog

#113

Beware! This problem is especially nasty when the system contains PC-compatible chipsets due to the extremely rapid evolution of desktop PC technology! The half-life of a PC chipset is about 3 Comdexes, where 1 Comdex = 6 months.

In a modular, PC/104-based system, you are buffered from having to struggle with individual IC obsolescence problems. When a particular IC on a PC/104 module is unavailable, your module supplier should provide you with an equivalent substitute module. If not, you always have the option of locating an alternate module somewhere else that performs a similar function. Hopefully, an obsolete IC will never force you to a board-level redesign.

KEEP YOUR OPTIONS OPEN!

If you want to take full advantage of the flexibility that a PC/104-based system design can offer for future options, upgrades, and substitutions, you *must* treat each PC/104 module as a generic function block.

Why?

This modularity ensures that you can substitute equivalent modules for the ones you must replace (rest assured, you will need to replace some eventually!). However, there are some specific guidelines for increasing modularity.

Avoid using the chip-specific features of PC chipsets. Unless a particular function in a PC chipset is part of the PC standard (or at least part of a well-defined and multiple-sourced superset), fight the temptation to use it! By building your application on generic functionality, the system you design is protected from component obsolescence through module-level substitution. Only a system based on generic PC/104 function blocks readily offers alternate sourcing of modules, high- and low-performance substitutions, and future backwards-compatible migration paths.

Wrap software drivers around nonstandard functions. Despite the desire to keep things generic, there are times when you need to use functions that aren't part of the normal PC standard. In these situations, it's important to keep a software layer between the application program and the nonstandard hardware.

With this object-oriented hardware and software approach, you have the flexibility of being able to alter hardware without rewriting the main application code. This is true as long as .the hardware differences are adequately masked by an intervening software layer.

For this reason, try to select PC/104 modules that come with BIOS or software drivers for all nonstandard hardware functions. This assures your ability to maintain a common function set despite future hardware changes that may be required or desired.

CONCLUSION

PC/104 embedded-PC modules offer highly efficient building blocks for designing embedded systems using the popular and user-friendly IBM-PC architecture. With more than 140 vendors offering off-the-shelf PC/104 modules and additional hardware and software vendors announcing PC/104 products nearly every week, we can expect to see PC/104 in an increasing number of embedded systems for at least another decade.

Consider using PC/IO4 modules to create a flexible object-oriented hardware architecture for your next embedded system project, as an alternative to the traditional embedded microcontroller approach in which you completely "reinvent the wheel" for each project! ❑

Rick Lehrbaum cofounded Ampro Computers where he served as vice president of engineering from 1983–1991. Now, in addition to his duties as Ampro's vice president of strategic programs, Rick chairs the PC/104 Consortium and the IEEEP996.1 working group, which is developing an IEEE version of PC/104. He may be reached at rickl@ampro. corn.

ABOUT THE PC/104 CONSORTIUM

February 1991, the nonprofit PC/104 Consortium was formed with the objective of maintaining and distributing the PC/104 *Specification* and publishing listings of PC/104 products and vendors. Its membership now numbers over 135 companies, all of which offer PC/IO4 modules and related goods and services.

There are no licenses or fees required to use PC/104. Users and manufacturers of PC/104 modules do not need to be members of the PC/104 Consortium. However, Consortium members gain the use of the PC/104 logo and are included in the PC/104 *Resource Guide* as well as other company and product listings.

For further information on PC/104, see the contact information above.

SOURCES

PC/1 04 Specification, PC/1 04 Resource Guide, and PC/1 04 Product Index
PC/104 Consortium
P.O. Box 4303
Mountain View, CA 94040
(415) 9038304
Fax: (415) 967-0995
Fax on demand: (408) 720-0515

IEEEP996 Draft Specification
IEEE Publications
(908) 981-1393

Using the PC Architecture in Embedded Applications
Ampro Computers, Inc.
990 Almanor Ave.
Sunnyvale, CA 94086
(408) 522-2100
Fax: (408) 720-1305

The XT/AT Handbook
Annabooks
(619) 673-0870
Fax: (619) 693-1432

407 Very Useful
408 Moderately Useful
409 Not Useful

An LCD and Keypad Module for the SPI

FEATURE ARTICLE

Brian Millier

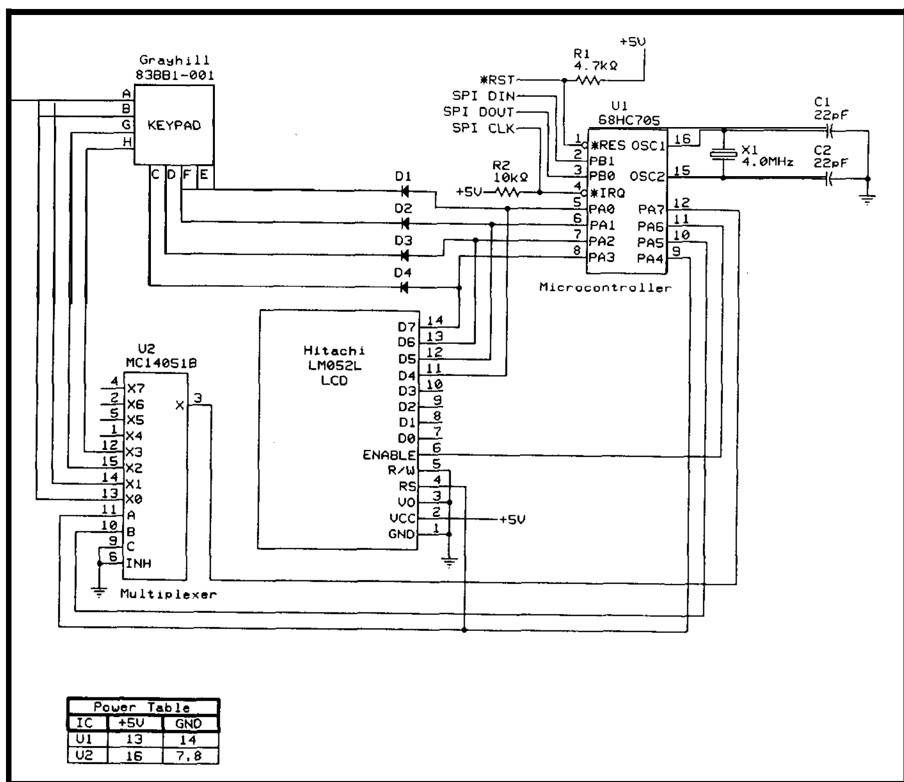
Appalled at the cost of a 74C922, Brian designs a circuit to connect a keypad and LCD module to microprocessor boards. The serial peripheral interface (SPI) provides fast data transfer with little code support.

One the other day, while placing an order for 74C922 keyboard encoder ICs, I thought back to the early '70s when *Popular Electronics* featured a construction article entitled "The Cosmac Elf." The article described an early personal computer based on RCA's 1802 CMOS microprocessor.

I built and used one of these computers at the time. While the RCA 1802 microprocessor never became popular, the 74C922 keyboard decoder used in this project is still alive and kicking. Although I was intrigued with the CMOS chip at the time, I am shocked when I now order them at a cost of $11 (Canadian currency). Why, for that price, I can get a complete microcontroller chip with onboard EPROM, RAM, timers, and more!

Instead of paying the big bucks, I designed a simple circuit to replace the 74C922, which offered a simple interconnection of both a keypad and an LCD module to commercial microprocessor boards. While it is not difficult to interface a keypad and LCD module to a micro, there are a few pesky design problems to overcome:

- Using the 74C922 encoder requires access to the data bus as well as a device-select signal and input port or interrupt pin for the data available signal. Alternatively, a spare parallel port may be used if one is available.
- The LCD module requires an ENABLE signal. Since device-select signals may be hard to come by on a small board totally populated with RAM and EPROM, you may need an additional 74LS138 decoder IC. Since the LCD needs an active-high select and most chip-select signals are low, toss in an extra 74LS04 inverter.



Power Table		
IC	+5V	GND
U1	13	14
U2	16	7,8

Figure 1—*The SPI-based* LCD/keypad circuit is based on a *68HC705* microcontroller and *LM052L* LCD display module. *Most any* **matrix** *keypad can be used.*

Listing 1-Code to set up *fhe SPI on a 68HC11 and make use of fhe LCD/keypad module is minimal. This code was assembled using the freeware AS1 1 assembler.*

```
DDRD     equ   $1009
SPCR     equ   $1028
SPSR     equ   $1029
SPDR     equ   $102A

sci_rec  equ   $FFCD        * BUFFALO 3.4 monitor SCI routines
sci_send equ   $FFAF

         org   $2000
start
         jsr   initSPI      * Initialize the SPI port
         ldaa  11127
         jsr   SPItrans     * This command is sent twice, as 1st
                            * byte received after power up may
                            * be misread
         ldaa  #127         * Set KBDLCD module to COMMAND mode
         jsr   SPItrans
         ldaa  #001         * Send an "LCD Clear" command
         jsr   SPItrans

* LCD needs more time to process the CLEAR command
* than for displaying chars to the LCD screen
         ldx   #1000
al       dex
         bne   al

         ldaa  #126         * Set KBDLCD module to DATA mode
         jsr   SPItrans

         ldx   #message     * Point to the ASCIIZ message string
sl       ldaa  0,x
         beq   s0
         inx
         jsr   SPItrans     * Send out string, 1 char at a time
         bra   sl

* LOOP: take a char from SCI in, send it to KBDLCD, and
* send the KBDLCD key pressed code back to host via SCI
so       jsr   sci_rec
         jsr   SPItrans
         jsr   sci_send
         bra   s0

message  fcc   'This is a test!'
         fcb   0

* Send a byte in A out SPI, and return with rcvd SPI byte in A
SPItrans
         staa  SPDR
s2       ldaa  SPSR
         bita  #$80
         beq   s2
         ldaa  SPDR
         anda  #$7f         * Take binary keycode 0-16
         adda  #$30         *  and bias it into ASCII range

* Allow a short delay time for KBDLCD module
* to process the character to the LCD
         ldab  #20
s3       decb
         bne   s3
         rts

* Initialize the SPI at slowest rate: 62.5 kbps
initSPI
         ldaa  #$5f
```

(continued)

- The typical bus-cycle time of most common microcontrollers is shorter than that called for by the LCD manufacturer. I have generally found them to work, but there are no guarantees.

I decided to make use of the SPI port, which exists on most microcontrollers and is often unused. Using this high-speed serial link and a Motorola 68HC705K1P microcontroller, I have designed a very simple LCD and keypad interface which uses only the three SPI signals from the host controller. The serial port is also handy if the operator's panel is far away from the microcontroller circuit board itself.

The cost of the circuit is little more than the 74C922 which it replaces. I chose the Motorola microcontroller since I've had their nifty $50 evaluation board and software for a year or so now, and finally the 68HC705K1P is available. I expect that the 16C54 PIC family chips would also serve my low-cost purpose, but I have more experience programming the Motorola family.

MAKING THE CONNECTION

For this circuit to be generally useful, it must offer fast data transfers to the LCD display and require very little code support in the host microcontroller. The Serial Peripheral Interface (SPI) available on the 68HC11, TMS370, and some of the 8031 derivatives, satisfies both these criteria. If you are not familiar with this functional block, refer to the SPI sidebar for a brief overview of Motorola's implementation of it.

Readers familiar with the Motorola 68HC705K1 family are likely saying, "Whoa-there is no SPI circuit block in that chip"-which is correct, of course. The trick used in this design implements the SPI in software. I wanted this circuit to work with SPI ports on at least the two microcontrollers that I commonly use-the 68HC11 and the TMS370.

Of the two, the 68HC11 is much less programmable in terms of bit rate. Its slowest SPI bit rate (with a 2-MHz E clock) is 62.5 kbps or 16 us per bit.

Getting that timing right is a critical aspect of the code for the 68HC05KI (as I will cover fully later).

It takes 128 us to transfer a data byte to the LCD over the SPI. Since the LCD needs about 50 µs between each character it receives, this circuit doesn't slow things down too much. Reasonably rapid LCD screen updates are possible. As well, the keypad data is returned to the host at the same 62.5-kbps rate, although since the keyboard-input functions are slow, this is not a concern.

Code overhead on the host micro is minimal. Listing 1 presents the short routine for initializing the 68HC11 SPI. It is important to note that the 68HC11 SPI block is set up for a clock phase and polarity of 1 since this is the only way this circuit will work! The code to send a message to the LCD plus perform other functions is also shown in Listing 1. Since the BUSY signal of the LCD is not read by the KBDLCD module, software loop delays are built into the routines so the KBDLCD module can keep up.

Although it is not shown in the program code, it is very important to remember to tie the 68HC1 l's -SS line to Vcc to make the 68HC11 the master device. If you're using a microcontroller without an SPI port, it would not be too hard to write a bit-banging routine to implement the SPI using three I/O port lines. It would, however, have a slower data transfer rate unless the processor was very fast.

THE HARDWARE

The entire circuit is detailed in Figure 1. Apart from the crystal and its capacitors, the 68HC705K1P needs nothing else other than 5 V to run. The reset function is looked after internally by the chip's timer subsystem. However, this internal reset circuit releases the processor from reset after 4064 clock cycles. V_{DD} must be stable by this time or correct operation will be uncertain. If your power supply does not come up to spec quickly enough, connect the 68HC705 -RESET line to the host -RESET line.

The 16-button keypad can be any 4 x 4 matrix such as a Grayhill 83BB1-001 or the DMC DS16 membrane

Listing *I-continued*

```
        staa  SPCR         * Enable SPI as Master CPOL,CPHASE=1,
clk/32
        ldaa  #$18
        staa  DDRD         * MOSI,SCK made outputs
        ldaa  SPSR         * Clear flags
        ldaa  SPDR
        rts
```

Listing 2—*The SPI data-transfer interrupt-service routine for 68HC705K1 uses straight-line code in the interest of speed.. It was assembled using the P&E IASM05K assembler.*

```
        org    RAM
rxchar  rmb    1
txchar  rmb    1
flag    rmb    1
        org    ROM

* Mainline program code here

* SPI data transfer Interrupt Service Routine
IROISR
        clr    rxchar        ; Ignore first MSB for lack of time
isrl    bil    isrl
isr2    bih    isr2          ; Wait until next falling edge
        lda    txchar
        sta    prtb          ; Send out bit 7
        lsr    txchar        ; Shift for next time
isr3    bil    isr3          ; Wait until rising edge
        brclr  1,prtb,isr4
        bset   6,rxchar
isr4    bih    isr4          ; Wait for falling edge
        lda    txchar
        sta    prtb          ; Send out bit 6
        lsr    txchar        ; Shift for next time
isr5    bil    isr5          ; Wait until rising edge
        brclr  1,prtb,isr6
        bset   5,rxchar
isr6    bih    isr6          ; Wait for falling edge
        Ida    txchar
        sta    prtb          ; Send out bit 5
        lsr    txchar        ; Shift for next time
isr7    bil    isr7          ; Wait until rising edge
        brclr  1,prtb,isr8
        bset   4,rxchar
isr8    bih    isr8          ; Wait for falling edge
        lda    txchar
        sta    prtb          ; Send out bit 4
        lsr    txchar        ; Shift for next time
isr9    bil    isr9          ; Wait until rising edge
        brclr  1,prtb,isr10
        bset   3,rxchar
isr10   bih    isr10         ; Wait for falling edge
        lda    txchar
        sta    prtb          ; Send out bit 3
        lsr    txchar        ; Shift for next time
isrll   bil    isrll         ; Wait until rising edge
        brclr  1,prtb,isr12
        bset   2,rxchar
isr12   bih    isr12         ; Wait for falling edge
        lda    txchar
        sta    prtb          ; Send out bit 2
        lsr    txchar        ; Shift for next time
isr13   bil    isr13         ; Wait until rising edge
```

(continued)

Listing *P-continued*

```
        brclr  1,prtb,isr14
        bset   1,rxchar
isr14   bih    isr14          ; Wait for falling edge
        lda    txchar
        sta    prtb           ; Send out bit 1
isr15   bil    isr15          ; Wait until rising edge
        brclr  1,prtb,isr16
        bset   0,rxchar
isr16   lda    #82h
        sta    ISCR           ; Clear IRQ flag since IRQ latch will
                              ; be set from 7 SPI clocks following
                              ; the initial one which caused this
                              ; ISR to be invoked
        clr    txchar         ; Zero out txchar
; Additional code to send byte to the
; LCD display, in two 4-bit nybbles
        rti
***************************************************
        org    vectors           ; Vectors begin at $03F8
        dw     rom
        dw     IRQISR            ; -IRQ vector
        dw     rom
        dw     start             ; Reset vector
```

keypad that I used (see Photo 1). The columns are scanned by sequentially placing a high level on PAO-PA3. The rows are sensed by sending four binary combinations to PA4 and PA5, which are connected to the address inputs of the 4051 multiplexer chip.

The chip then routes each row in sequence to the PA7 port, configured as the sense input. The A port has programmable pull-down circuitry for any port bits configured as inputs, so no additional resistor is needed. Diodes D1–D4 prevent the possible shorting of two PAO-PA4 data lines should the operator hold two keys pressed at one time. This would only be a problem if the LCD was being

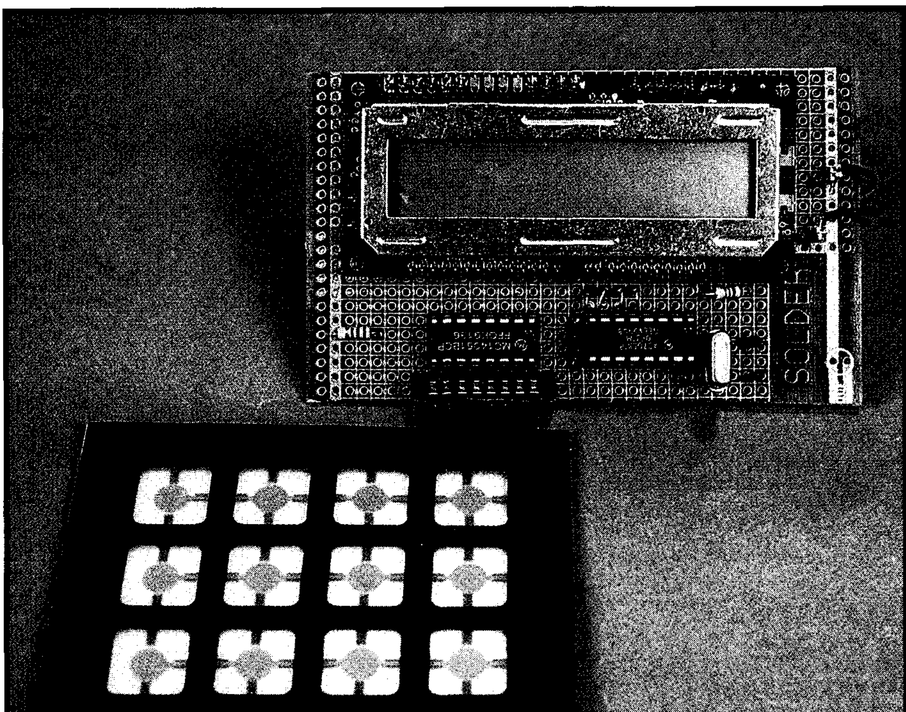
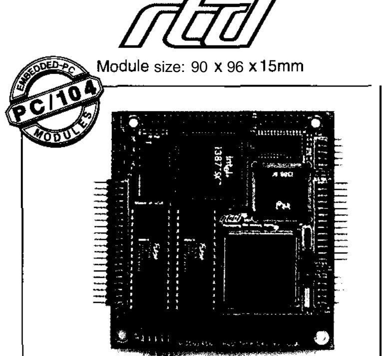


Photo 1—*The complete SPI LCD/keypad circuit is built on a small perfboard and uses a DMC DS16 membrane keypad.*

REDUCE THE STACK!
Use fully integrated PC/104 CPU and DAS modules from rtd®

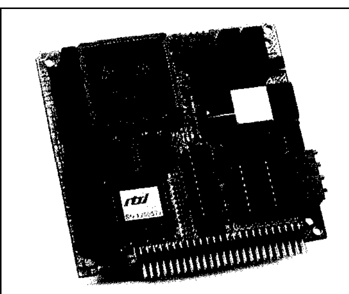
Module size: 90 x 96 x 15mm



PC/104 Compliant PC-AT SBC
CMi386SX-1: $578 /100 pieces
2MB DRAM & SSD software included

5 **PC/XT/AT Single Board cpuModules™:**
- 486SLC, 386SX, F8680, V41 & VG230 DOS CPUs
- 80387SX math coprocessor socket on-board
- 512KB, 1 MB, 2MB or 4MB DRAM installed
- Two 32-pin SSD sockets & support software
- IDE, floppy & CGA controllers
- RS-232/422/485 serial ports
- Bidirectional parallel, keyboard & speaker ports
- Keypad scanning & PCMCIA support
- Power management & single +5V supply

7 **utilityModules™:**
- Super VGA controller & I/O modules
- PCMCIA carriers for Types I, II & III cards



PC/104 Compliant 200 kHz Analog I/O Module
DM5408-2: $498 /100 pieces

17 DAS **dataModules®:**
- 12 & 14-bit A/D conversion up to 200 kHz
- Gap free, high speed sampling under Windows™ & DOS
- Programmable scan, burst & multiburst
- Pre, post & about triggers
- 1 K channel-gain scan memory with skip bit
- 1024 sample A/D buffer
- 12-bit analog outputs
- Bit programmable digital I/O with Advanced Digital Interrupt modes
- Incremental encoder interfaces
- 4-20 mA current loop source
- opto-22 compatibility
- Low power & single +5V power supply

For *technical specifications and data skeets on PC/104, ISA bus and Eurocard products, call*
RTD USA Technical **FaxBack:** 1 (814) 235-1260
RTD USA BBS: 1 (814) 234-9427

rtd® **Real Time Devices USA**
200 Innovation Boulevard • P.O. Box 906
State College, PA 16804-0906 USA
Tel: 1 (814) 234-8087 • Fax: 1 (814) 234-5218

RTD Europa RTD Scandinavia
Fax: **(36) 1 212-0260** **Fax: (356) 0 346-4539**

RTD is a **founder of the** PC/104 Consortium and the world's leading supplier of PC/104 CPU and DAS modules

#115

updated at the same time, but it is a possibility. The necessary debounce of the switches is done in software.

The LCD module can be any one of many inexpensive LCD modules available on the surplus market that use the Hitachi HD44780 LCD controller driver. I am using the Hitachi LM052L, a 16-character by 2-line module. With so few port lines available on the 68HC705K1, I had to use the 4-bit transfer mode. From the host micro's standpoint, the LCD is sent data as if it were an 8-bit device.

Port bits PAO-PA3 serve double duty as the data bus for the LCD and as the keypad-scan function described earlier. Port PA4 also serves double duty as the register-select signal for the LCD and keypad column-multiplex address. PA6 is the LCD ENABLE strobe signal. During the key scan, ENABLE remains low so the LCD does not receive extraneous data.

SPI data comes in to PB1 and is sent out on PBO. The SPI clock signal is connected to the –IRQ input. Note that the –IRQ line is pulled high by a 10-kΩ resistor. This ensures the 68HC705K1 is not stuck in an extended interrupt prior to the host micro's proper initialization of the SPI port pins. The power consumption of the entire circuit, using a 16 x 2 LCD, is 26.5 mW.

THE FIRMWARE

Steve Ciarcia has stated, "My favorite programming language is solder." And, like him, I love building circuits.

At times in the past, I have happily wired complicated microprocessor circuits, assuming that I could write the necessary software later. When I reach the software and firmware stage, I am sometimes dashed by the realization that I neglected to investigate software considerations such as critical timing.

In this case, I knew from the outset that getting the 68HC705K1 to handle the SPI data transfers at 62.5 kbps was going to be tricky. I was also concerned whether 496 bytes of EPROM was going to be enough, although I haven't yet written an assembly language program that was

Listing *3-The 4-bit LCD routines for the SPI LCD/keypad module are written for the 68HC705K1 microcontroller.*

```
*lcd_clr-clears display without   regs
* lcd_init-initializes LCD device. Invoke first.
* lcd_write-writes char in A at current cursor position

temp        rmb     1
lcd_init    lda     #$03
            sta     prta
            bset    6,prta
            bclr    6,prta
            jsr     lcd_dlay
            lda     #$03
            sta     prta
            bset    6,prta
            bclr    6,prta
            jsr     lcd_dlay
            lda     #$03
            sta     prta
            bset    6,prta
            bclr    6,prta
            jsr     lcd_dlay
            lda     #$02
            sta     prta
            bset    6,prta
            bclr    6,prta
            jsr     lcd_dlay
            lda     #$28
            jsr     lwritec
            jsr     lcd_dlay
            lda     #$0e
            jsr     lwritec
            jsr     lcd_dlay
            rts

lcd_clr     lda     #$01
            jsr     lwritec
            jsr     lcd_dlay

lcd_wait    sta     temp
            lda     #$60
lcdwl       deca
            bne     lcdwl
            lda     temp
            rts

lcd_write   jsr     lwrited
            jsr     lcd_wait        wait 50 µs for LCD to finish
            rts

* 5 ms delay
lcd_dlay    sta     temp
            clra
lcddl       inca
            deca
            inca
            deca
            inca
            deca
            inca
            deca
            inca
            deca
            deca
            bne     lcddl           256*36* 1/2 = 4.6 ms
            lda     temp
            rts
```

(continued)

Listing 3—*continued*

```
* write a byte to LCD command register
lwritec    sta     temp
           lsra
           lsra
           lsra
           lsra
           sta     prta            ; RS line low
           bset    6,prta          ; strobe ENABLE
           bclr    6,prta
           lda     temp
           sta     prta
           bset    6,prta          ; strobe ENABLE
           bclr    6,prta
           rts

* write a byte to LCD data register
lwrited    sta temp
           lsra
           lsra
           lsra
           lsra
           ora     #$10            ; RS line high
           sta     prta
           bset    6,prta          ; strobe ENABLE
           bclr    6,prta
           lda     temp
           ora     #$10            ; RS line high
           sta     prta
           bset    6,prta          ; strobe ENABLE
           bclr    6,prta
           rts
```

too big for the EPROM space I had available.

Therefore, I first designed the SPI transfer part of the program, calculated its timing, and when satisfied it would work, built the circuit. I am hoping that some of this methodology rubs off onto future projects!

Listing 2 shows the SPI interrupt service routine. The SPI data handling is performed using a pseudointerrupt technique. That is, the SPI clock (from the host micro), connected to the –IRQ line, generates an interrupt for the first clock (of an SPI transfer) received. This interrupt is necessary to ensure that the 68HC705 is always ready to receive a byte of SPI data.

However, the interrupt latency is 10 cycles plus the time it takes to finish the instruction being executed. This is too long a period for the 68HC705 to send and receive a bit.

The trick is to settle for 7-bit transfers, which are suitable for the LCD. The keypad-data output needs only 4 bits. The first SPI clock invokes the interrupt-service routine, does some housekeeping, then enters a polling loop to wait for the next SPI clock. The first SPI data bit (MSB) is simply ignored. The 68HC705 has BIH and BIL instructions for tightly polling the IRQ pin level. Using these instructions and tight, replicated, inline code, the program can both send and receive the SPI data with no problems.

Since SPI data is sent MSB first, the transmitted byte (keypad data) must be bit reversed before being sent. There is plenty of time to do this bit reversal during the keyboard-scan routine using a 16-entry lookup table.

I should note that it is critical that the IRQ flag-clearing instruction be included at the end of the ISR. During the execution of this ISR, seven additional falling clock edges have been applied to the –IRQ line, and its latch will definitely be set. Without the flag-clearing instruction, the ISR would be reentered even though the SPI data byte has been fully sent and received.

The circuit returns values of 1-16 to the host for the 16 different keys. A

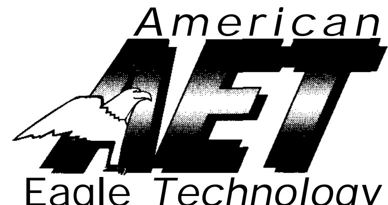
A Complete Family of Data Acquisition Products from

American Eagle Technology

- Data acquisition rates from 100kHz to 1 MHz.
- Newest designs incorporate the latest technology: FIFO buffers, dual-DMA, REP INSW data transfers, programmable gains, etc.
- Simultaneous sampling option for all boards.
- Lowest prices on the market for comparable performance.
- Digital I/O w/ on-board relays.
- 16-bit analog output (D/A).
- Large inventory of popular items for next day delivery.
- Also available: signal conditioning, multi-port serial boards, device programmers, logic analyzers, & much more.

Only American Eagle Technology Gives You All This Free Software:

- *WaveView* menu-driven software
- *Discstream* high-speed streaming software
- Complete software developers' kits for both DOS and Windows
- Drivers for all popular application programs such as LabView, LabWindows, NOTEBOOK, SnapMaster and DASyLab

American


Eagle Technology

526 Durham Rd
Madison, CT 06443

Call: (203) 245-6133
Fax: (203) 245-6233

Send for Your FREE Catalog Today!

#117

value of zero is sent when no keys have been pressed since the last inquiry from the host.

Remember that the 96-character ASCII set can be transmitted to the LCD using seven bits. To send data to the command register for such operations as display clearing, I have "stolen" three seldom-used character values (125-127) for that purpose. The definitions of these three commands are as follows:

Cursor mode (125)—the next byte(s) sent move the cursor to the requested position
Data mode (126)—the next byte(s) sent go to the data register of the LCD
Command mode (127)—the next byte(s) sent go to the command register of the LCD

All necessary commands to the LCD are seven bits long with the exception of the cursor-movement command, which has an additional mode assigned to it. All values passed to the LCD while in this mode have DB7 set high as required for cursor movement.

The necessary code (with attendant timing constraints) to set up the LCD display properly in 4-bit mode executes when the 68HC705 is powered up. The host micro need not worry about this other than to wait 15 ms or so after reset before sending any

Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is a high-speed, TTL-compatible, full-duplex, synchronous, data-transfer protocol. It has been implemented as a functional block on many modern microcontrollers including the 68HC11, TMS370, and some derivatives of the 8051 family. While they're not all that common, there are peripheral ICs designed for this serial bus.

For instance, Maxim and Linear Technology make multichannel 12-bit ADCs. Texas Instruments makes ADCs and power driver ICs (these make excellent stepper motor drivers). Motorola's LCD driver ICs use a serial interface, which can be driven by the SPI. These are just a few examples of available devices.

The SPI functional block, which I outline, is implemented by Motorola in their 68HC11 family. There are slight differences in the implementation of this protocol in other manufacturers' products, but the principle is the same.

Data transfer is serial and, unlike the more common RS-232 standard, is synchronous. Since this bus was originally intended to provide communication among microcontrollers in a multiprocessor environment, the concept of a master and slave is used.

In environments, where both devices on the bus are "intelligent" and therefore capable of originating messages on their own, the protocol allows only *one* of the devices to be the master. So, only the master can initiate messages; the slave receives and responds to these messages. In a project such as this one, the host microcontroller must be programmed as the master; the keyboard and LCD module is the slave. (Program code and implementation are detailed in the article.)

The SPI uses three signals to transfer the data: MOSI (Master Out Slave In), MISO (Master In Slave Out), and CLOCK. Since the 68HC11 is the host or Master processor, the MOSI line carries data to the LCD and the MISO carries data from the keypad to the host [they would be reversed if the 68HC11 were the slave).

The `initSPI` procedure in Listing 1 gives the correct sequence of 68HC11 instructions for setting the SPI properly. Note also that the –SS line of the 68HC11 must be tied to **Vcc** to enable it as the master.

The CLOCK signal needs further explanation. The rate of data transfer depends on the CLOCK rate. In the 68HC11, the clock rate is based on the processor clock divided by the constants 2, 4, 16, or 32. The standard processor clock rate for the 68HC11 is 2 MHz. This project uses the /32 option, as the 68HC705K1 in the keyboard and LCD module cannot respond to higher rates. This selection provides a transfer rate of 62.5 kbps. Incidentally, the TMS370 provides a much more programmable SPI clock-rate selection with a divisor ratio of up to 1024 on its internal clock.

The 68HC11SPI block offers both the clock polarity and phase to be programmed. Through this, the SPI works with peripheral ICs made by many manufacturers. Due to the way in which the 68HC705K1 firmware works, it is important that both the clock phase and polarity be set to 1 for this project.

The SPI is a full-duplex protocol. Unlike other common, full-duplex protocols such as the RS-232 where there is not necessarily a 1:1 relationship between the amount of data sent and that received, the SPI does impose this constraint. For every byte sent out by the master, a byte is simultaneously clocked in to the master. Whether the slave actually sends back data is immaterial. The master assembles a byte of data from the signal seen on its MISO line during the time its data byte is being sent out.

For the purpose of sending data from the master to an output-only peripheral such as a DAC, this incoming byte would be ignored. In the case of an SPI device such as an ADC, which must be triggered and then read, the common method is for the ADC to return the last reading it took at the time that it is receiving its trigger command for the new conversion. In this project, the module returns the last key pressed whenever it receives an incoming LCD data byte.

The 68HC11 SPI block, while very flexible, has a fixed 8-bit word length. The TMS370 SPI block has a fully programmable word length of 1–8 bits. While the SPI is certainly not as flexible as the I²C bus, it is much easier to use when only a small number of devices need to be connected together.

data to the LCD and keypad circuit. For those who want to use these LCD modules in 4-bit mode in their own applications, I have shown the required code in Listing 3. The Hitachi HC44780 data manual is a comprehensive source of information on programming these modules.

So the builder can see if the circuit is working properly prior to being connected to a host SPI port, the 68HC705 sends out the sign-on message "CIC SPI LCDKBD" after it initializes the LCD module. Once connected to a host, the first LCD command should be to clear the LCD of the sign-on message.

To read the keypad, the host sends a byte to the SPI. The key code is returned in the SPI data register with a zero indicating no key presses since the last poll. If you want to read the keypad without also writing to the LCD, send code 126, which sets up data mode (the mode most commonly used). Note that this circuit "remembers" the last key pressed since the last host polling. This feature ensures that if critical timing sequences are performed, the host is able to check the keypad less frequently.

WRAP-UP

I hope this article prompts anyone who hasn't bothered to make use of the SPI to give this simple, yet useful, circuit a try. It can be breadboarded in about an hour or so. ❑

Brian Millier has worked as an instrumentation engineer for the last 12 years in the Chemistry Department of Dalhousie University, Halifax, NS, Canada. In his leisure time, he operates Computer Interface Consultants and has a full electronic music studio in his basement. He may be reached at brian.millier@dal.ca.

SOURCES

LCD module
Timeline Inc.
23650 Telo Ave.
Torrence, CA 90505
(3 10) 7845488

MC68HC705K1P
Jameco Ltd.
1355 Shoreway Rd.
Belmont, CA 94002-4100
(415) 592-8097

Motorola Technical Manuals
Motorola Literature Distribution
P.O. Box 20912
Phoenix, AZ 85036.
(602) 244-6900

A programmed 68HC705K1P is available for $15 plus $3 postage and handling (U.S. currency) from:
Brian Millier
Computer Interface Consulting
P.O. Box 65, Site 17, R.R. 3
Armdale, NS
Canada B3L 4J3
(902) 876-8645
E-mail: brian.millier@dal.ca

I R S

410 Very Useful
411 Moderately Useful
412 Not Useful

STOP
LOOK
LISTEN!

Odds are that some time during the day you will stop for a traffic signal, look at a message display or listen to a recorded announcement controlled by a Micromint RTC180. We've shipped thousands of RTC180s to OEMs. Check out why they chose the RTC180 by calling us for a data sheet and price list now.

MICROMINT, INC.
4 Park Street, Vernon, CT 06066
(203) 871-6170•Fax (203) 872-2204

in Europe: (44) 0285-658122•in Canada: (514) 336-9426•in Australia: (3) 467-7194•Distributor Inquiries Welcome

#119

BONUS SECTION

Home Automation & Building Control

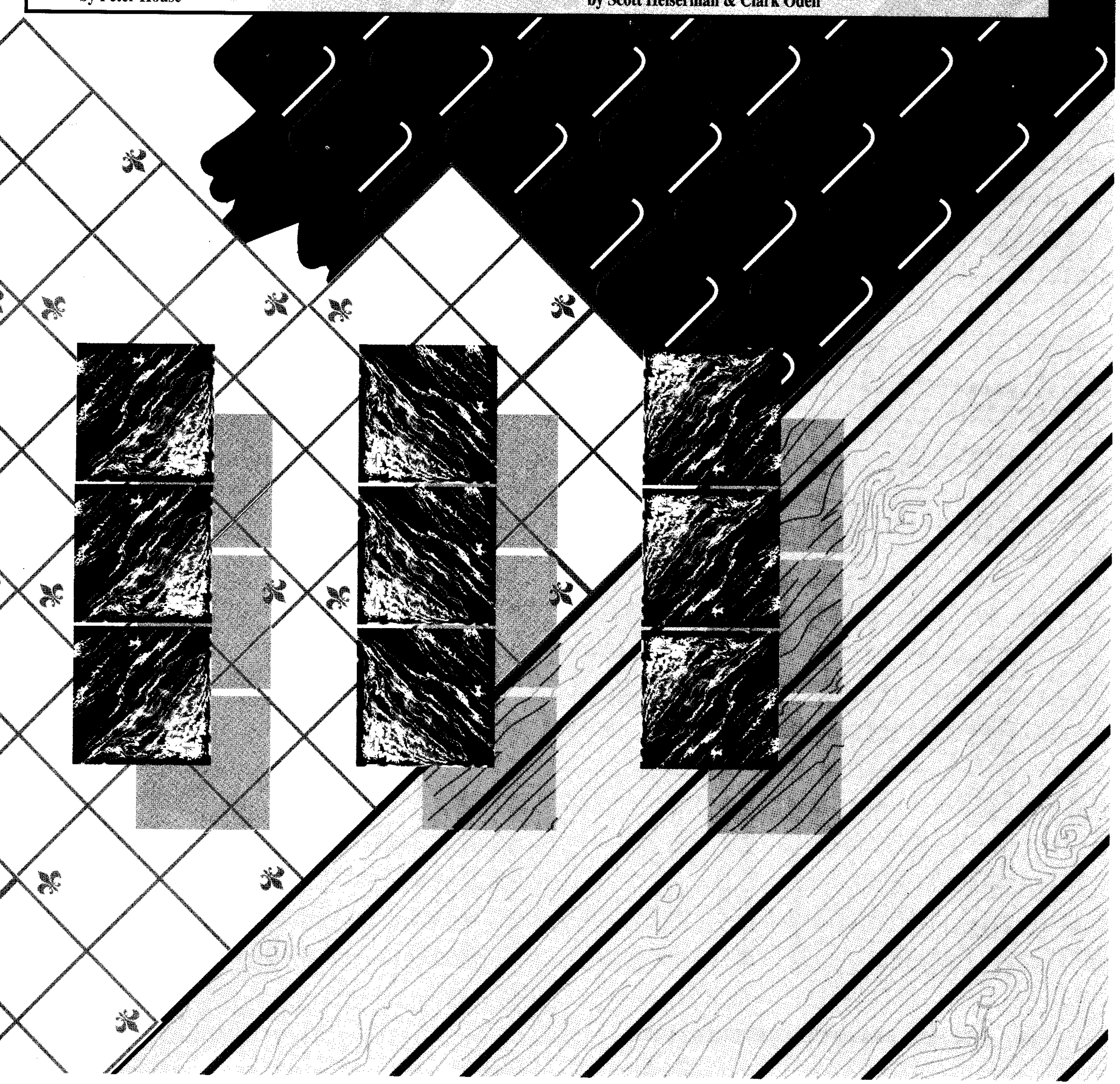
Page 49 Innovations in Home Automation & Building Control

51 Developing Home Automation Devices with LonWorks
by Rich Blomseth

61 CEBus for the Masses
by Peter House

69 The Blind Robot: An X-10 Miniblind Automation System
by Herbert McKinney, Jr.

75 A Learning Remote-Controlled Speaker Selector
by Scott Heiserman & Clark Oden



IN HOME AUTOMATION & BUILDING CONTROL

edited by
Harv Weiner

PLUG-IN FREEDOM FOR THE DIGITAL HOME

An affordable, home-wiring system that fulfills consumers' immediate demands for home networking of computers, security systems, and other electronic products also offers a convenient on and off ramp to the coming information superhighway.

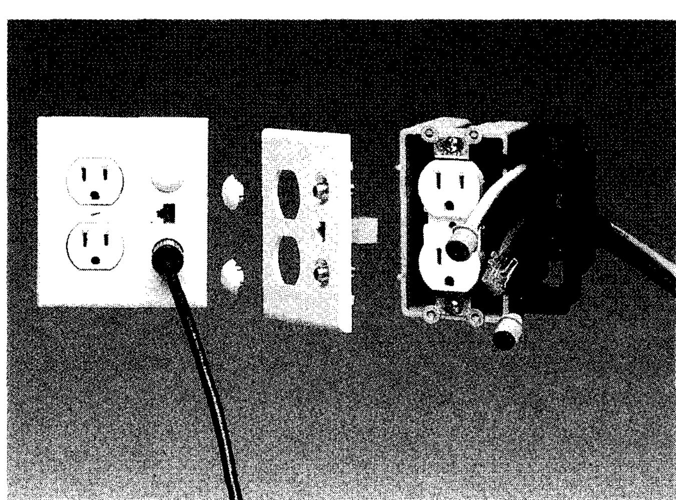
The new **TecSystem** from U.S.Tec is the backbone to an easy-to-use home LAN. Consisting of a wall plate (TecPlate), a central electronic server, and special networked cabling, the TecSystem allows homeowners to access cable TV, telephone, and electricity from a single, convenient wall source. Installed in multiple locations, the TecSystem enables plug-and-play flexibility with other electronic devices in the home. TecSystem is CEBus compliant.

The system's use of higher-bandwidth-capacity wiring prepares homeowners both for in-home automation of electronic products and appliances, and two-way access on the high-speed, high-volume digital superhighways. The TecSystem allows you to view VCR and security camera pictures on multiple TVs, network computers to printers, and send stereo audio from room to room. Home LAN is well-positioned for problem-free communication with global computing networks, programmable news and information, video on demand, multiple TV channels, and other multimedia services.

TecSystem comes in multiple configurations. An entry-level, four-TecPlate network can be installed for as little as $500. An eight-TecPlate network is priced at **$1500.** A complete network accommodates up to 32 TecPlates.

U.S.Tec
470 South Pearl St.
Canandaigua, NY 14424
(716) 396-9680
Fax: (716) 394-7095

#510



HIGH-POWER, PERMANENTLY WIRED X-10 LIGHTING MODULE

The PCS lighting control modules, **LM1-800**, **LM1-1200**, and **LM1-1800**, finally offer an economical solution to controlling more than 500 W of incandescent lighting using X- 10 signalling. These modules are mounted on a flat, vertical surface and are permanently attached to the residence wiring system. Two ½″ knockouts and a terminal block are provided for simple connection.

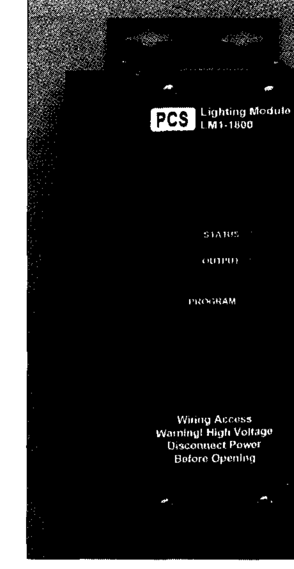
All controllers offer the same advanced features available on all PCS multimodules. These features are not available on any conventional lighting modules.

Lights can be brightened from full off without having to come to full on first. They also can dim down from full on. If dimmed past the lowest dim level, the module enters the full-off state, allowing it to go to full on with the next on command.

Another feature is that the current dim level is not lost if the module is turned off or if power fails. Each time the module receives an on command, it returns to the preset dim level, allowing the user to preset various indoor and outdoor lighting levels.

All versions of LM1 are thoroughly overdesigned with heavy-duty triacs, more than adequate heat sinking, EM1 protection, and all-metal enclosures. Modules are in the process of being UL listed.

All lighting modules can be optionally turned on and off by an external switch in series with the load, typically a standard wall switch. This is a convenient method of providing external manual control to every lighting circuit.



Powerline
Control Systems
9031 Rathburn Ave.
Northridge, CA 91325
(818) 701-9831
Fax: (818) 701-1506

#511

Innovations

IN HOME AUTOMATION & BUILDING CONTROL

XTEN-UTILITIES   REPORTS   (c) 1995 WCA

FILE .... : Events.Dat
HOUSE / UNIT SELECTED .. : ALL / 2   MATCHING EVENTS : 8

EVENT	HOUSE	UNITS 1 THRU 16	DAYS	TIME	FUNCTION	MODE
003	A	0100 0000 0000 0000	-T-----	12:15	ON	SECURITY
019	B	0100 0000 0000 0000	---T---	12:15	DIM TO 87%	SECURITY
035	A	0100 0000 0000 0000	-----S-	12:15	OFF	SECURITY
051	A	0100 0000 0000 0000	M------	12:15	ON	SECURITY
067	A	0100 0000 0000 0000	--W----	12:15	DIM TO 87%	SECURITY
083	A	0100 0000 0000 0000	----F--	12:15	OFF	SECURITY
099	A	0100 0000 0000 0000	------S	12:15	ON	SECURITY
115	A	0100 0000 0000 0000	-T-----	12:15	DIM TO 87%	SECURITY

SCROLL ... PRESS F3 for print menu ..... PRESS ESC TO EXIT

X-10 SOFTWARE FOR THE REST OF US

Wilmington Computer Applications has released a simple, nongraphics, menu-driven X-10 software package for use on most IBM PC, XT, or AT and Apple II computers. **XTen-Utilities** requires only one floppy, a serial port, 256 KB memory, and MS-DOS 2.0 (or newer) for IBMs. The Apple II version requires one floppy, super serial card (or equivalent), 64 KB memory, and includes ProDOS 8. Both versions require the CP290 X-10 home control interface.

XTen-Utilities offers file-based editing and reporting. A CP290 does not have to be connected while editing an event file or producing reports. Modified event files can later be uploaded to the CP290. The Reports command details house and unit usage. A countdown Timer function allows delayed X-10 device control. An adjustable Oscillate function can be used to cycle X- 10 controlled devices, including Power Horns, on and off.

Both versions include XTen-Menus. XTen-Menus allows easy two-keystroke control of any X-10 device and is easy to set up. You can even use your own descriptions on menus. Up to 16 submenus can be selected from the main menu. Any menu item can turn any one house code or multiple-unit combination on, off, dim, or flash.

XTen-Utilities is priced at $39.95.

Wilmington Computer Applications
P.O. Box 429 • Wilmington, MA 01887-0429
(SOS) 658-9950 **#512**

LOW-COST TOOL FOR DEVELOPING INTELLIGENT DEVICES

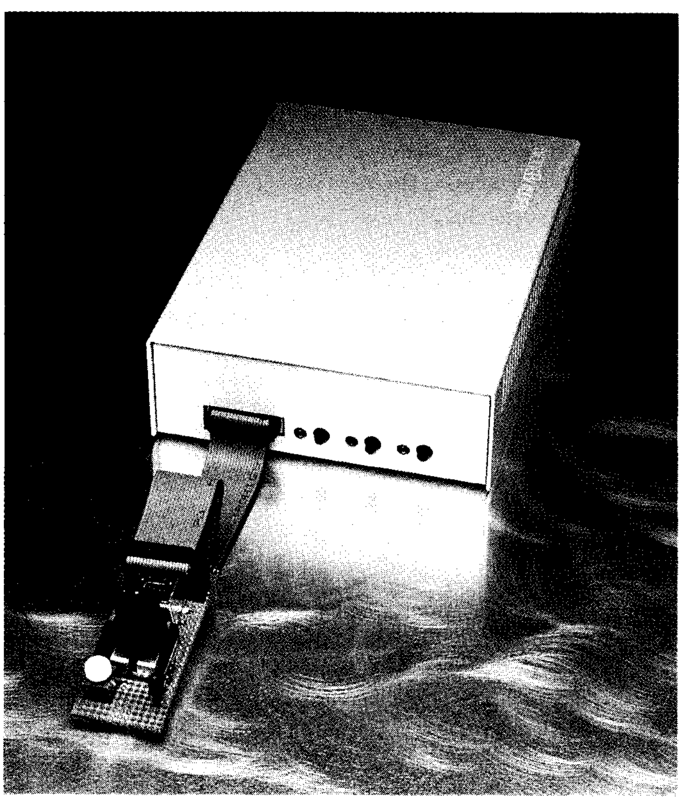
Echelon Corporation introduces **NodeBuilder**, a new development tool that makes it easy and inexpensive for manufacturers to design devices that can be integrated into automation and control networks. Installed LonWorks nodes today range from valves in chemical plants, to alarms in telephone central offices, to sensors for automated toll booths, to smart thermostats for homes.

LonWorks NodeBuilder includes everything developers need to create and test products for LONWORKS-based control networks. It uses a familiar Windows-based development environment with easy-to-use, on-line help. NodeBuilder includes the LonWorks Wizard, a tool which generates software for an interoperable LonWorks device.

NodeBuilder complements the development capabilities of the **LonBuilder Developers Workbench,** a tool with systems-level capabilities. System developers can use one or more LonBuilders for network development while simultaneously developing individual nodes for the system using LonWorks NodeBuilder.

The NodeBuilder is available for $3,995. For more information on LonBuilder Developers Workbench, call Echelon.

Echelon Corporation
4015 Miranda Ave.
Palo Alto, CA 94304
(415) 855-7400
Fax: (415) 856-6153 #513



Developing Home Automation Devices with LonWorks

RICH BLOMSETH

Rich believes **LonWorks** technology fills the gap between whole-house automation systems costing tens of thousands and the low-cost, **do-**it-yourself market. It provides a common device control scheme and communicates over media often already installed in the home.

Today, the market for home automation seems to be divided between whole-house automation systems costing tens of thousands of dollars and the low-cost, do-it-yourself market. What's missing is a systems-level approach providing the features of a whole-home system that is simple to install, is reliable, and comes at a low-cost.

LonWorks technology is Echelon's answer for that gap. It provides a method of communicating between devices using several types of media primarily for control. Although initially used mostly in industrial and commercial building control settings. LonWorks has become sufficiently popular that its prices have been driven down. It is now positioned for the low-cost home automation market.

Special codes embedded in each LonWorks device (e.g., a heater, thermostat, home theater center) are transmitted via the home's power lines (the technology is also available for twisted-pair, RF, infrared, coax, and fiber-optic media). To meet the desire for plug-and-play products. LonWorks provides a basic configuration which requires no installation or programming. For more customized automation, LonWorks devices can be programmed or integrated with other professional control systems.

After a brief introduction of LonWorks, this article will focus on the NodeBuilder, a development tool which enables engineers to create LonWorks devices.

WHAT IS LonWorks?

LonWorks technology is a system of sensors, actuators, displays, and logging devices (referred to as nodes) linked together to monitor and control electrical devices. Control functions are typically handled automatically, except for faults which the system cannot correct. In home automation

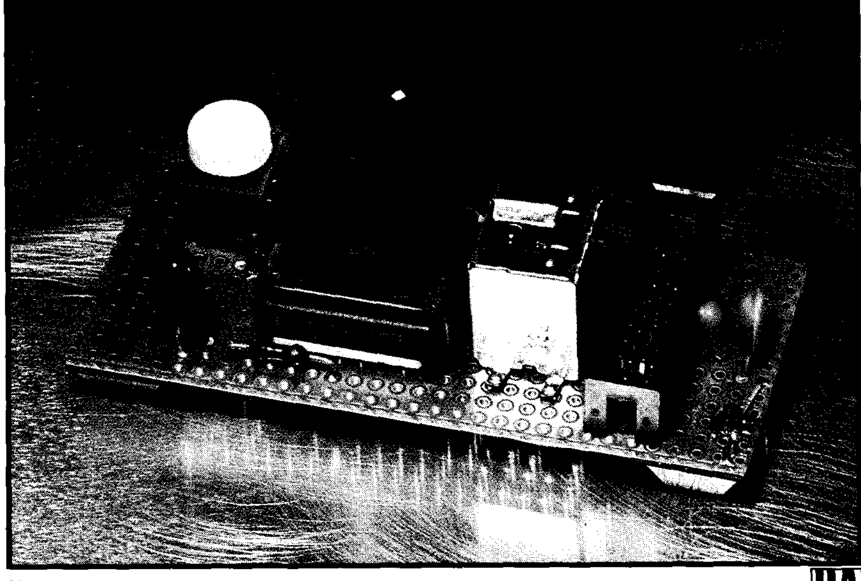


Photo **1:** *The prototype IR dimmer hardware was easily constructed using standard proto board and an IR receiver* **from** *Radio Shack.*

applications, a control network may provide safety (e.g., monitoring security, fire alarms, and pool and spa areas), control (e.g., regulating room temperature, lighting, draperies, and irrigation systems), and entertainment (e.g., managing A/V equipment).

Neuron chips, the heart of LONWORKS technology, contain the protocol (LONTALK) that enables them to communicate with other Neuron chips. Since Neuron chips can be connected directly to the sensors and outputs they supervise, a single Neuron chip handles processing of sensor and output status, execution of control programs, and communications with other Neuron chips.

For nodes requiring more processing or I/O power, the Neuron chip can also be used as a communications coprocessor for any other processor. The Neuron chip therefore provides a scalable solution that can be used even on complex nodes which include a host computer and network interface.

LONWORKS also provides interoperability with other control systems. Network management software, tools for installing complex networks, and routers enable communications between the different communications media.

RIDING THE POWER LINES

Power-line signaling is ideal for home-automation communications because it requires no new wires. As well, power wiring already reaches every device that needs to be controlled.

Although power-line signaling devices have been available for years, they have two significant drawbacks-they are unreliable and lack two-way communication. Intermittent noise sources, impedance changes, and attenuation conspire to make the power line a hostile path for power-line signaling.

To counteract these problems, LONWORKS combines narrowband signaling with signal processing and error correction algorithms in its transceivers. The transceiver features include:

- low-overhead error correction to enable the system to receive corrupted packets while maintaining a high throughput
- adaptive carrier-detect algorithm that automatically tracks changes in power-line noise levels
- impulse-rejection technology to improve performance in the presence of impulsive noise sources such as triac-controlled dimmers

Listing 1: *By declaring LONMARK objects and network variables for an IR dimmer,* any *device on the same network con communicate with the dimmer.*

```
#pragma set_node_sd_string "@1. IR Dimmer Controller"
network output sd_string("@0|1.")SNVT_switch nvoSwitch;
network output sd_string("@0|3.")SNVT_count nvoRawHwData;
```

I/O Device Name and Direction	Description
Bit, nybble, byte input and output	Direct binary I/O
Bitshift input and output	Up to 16 bits of clocked serial data
Dual slope input	Comparitor input for 16-bit dual-slope A/D
Edgedivide output	Waveform equal to fraction of input
Edgelog input	Edge to edge timing of an input stream
Frequency output	Square wave output of specified frequency
I²C input and output	Philips K-compatible serial I/O
Infrared input	Encoded input from an IR demodulator
Leveldetect input	Detect logic zero level
Magcard and Magtrackl input	IS0781 1 track 1 and 2 magnetic card readers
Muxbus input and output	Multiplexed address and data bus
Neurowire input and output	SPI and Microwire compatible serial I/O
Pulsecount output	Output specified number of pulses
Pulsewidth output	Output specified frequency and duty cycle
Oneshot output	Single output pulse of specified period
Ontime and Period input	Pulsewidth and period measurement
Parallel input and output	8-bit bidirectional I/O
Pulsecount and Totalcount input	Transition count over fixed or total interval
Quadrature input	Shaft encoder rotary position input
Serial input and output	8-bit asynchronous serial I/O
Touch input and output	Dallas Touch 1 -wire bus I/O
Triac output	Pulse delayed with respect to input edge
Triggeredcount output	Pulse controlled by counting input edges
Wiegand input	Wiegand card reader input

Table 1: *Built-in Neuron C I/O objects simplify interfaces to most common I/O devices.*

Listing 2: *IR dimmer software declarations for I/O objects configure the Neuron chip's internal hardware for the IR dimmer I/O devices.*

```
IO_0 output bit                          ioLED = 1;
IO_4 input quadrature                    ioDial;
IO_6 input infrared invert clock(7)      ioIRData;
IO_6 input bit                           ioIRDataLevel;
IO_7 input 1 eveldetect                  ioButton;
```

Listing 3: *The complete IR dimmer software listing shows how little code is requiredfor a complex application.*

```
// IRDIMMER.NC-Dimmer controller with manual and  nfrared
// inputs. Compatible with the Sony RM-V10 remote control.
// This remote puts out three (3) identical codes for each key
// closure.
//max_period = 2.6 ms; low bit= 1.1 ms; high bit.= 1.9 ms
//
// Object ID        Type
// 00               Switch sensor object, SNVT_switch

iipragma set_node_sd_string"@1. IR Dimmer Controller"
iipragma enabl e_io_pull ups
iipragma num_addr_table_entries 3
```

continued

listing 8: *continued*

```
// Open-Loop Sensor LonMark Object, ID #0
network output sd_string("@0|1.")     SNVT_switch nvoSwitch;
network output sd_string("@0|3.")     SNVT_count nvoRawHwData;
network input  sd_string ("@0|6.") config SNVT_count nciGain=5;

IO_0 output bit                          ioLED = 1;
IO_4 input quadrature                    ioDial;
IO_6 input infrared invert clock(7)      ioIRData;
IO_6 input bit                           ioIRDataLevel;
IO_7 input leveldetect                   ioButton;

// IR controller values
#define IR_ON_OFF 149
#define IR_VOL_UP 146
#define IR_VOL_DN 147

// ToggleSwitchState()-Toggle the state of the switch output.
void ToggleSwitchState(void){
  nvoSwitch.state = !nvoSwitch.state;
  io_out(ioLED, nvoSwitch.state ? 0 :1);


// ChangeSwitchLevel()-Change the switch level by a specified
// amount. Turn on the switch if the new level is not zero
// and the switch is off.

void ChangeSwitchLevel(long int deltaValue){
  long int tempValue;         // switch temporary update value

  tempValue = nvoSwitch.value + (deltaValue * nciGain);
  nvoSwitch.value = (unsigned) (tempValue < 0 ?
    0 : ((tempValue > 200) ? 200U: tempValue));
  if (nvoSwitch.value && !nvoSwitch.state)
    ToggleSwitchState0;


// Infrared data input task-Read data from infrared remote.
priority when(io_changes(ioIRDataLevel) to 0){
  unsigned int irData[2];                  // IR data

  if (io_in(ioIRData, irData, 12, 65424UL, 65424UL + 59UL) ==
     12) {
    nvoRawHwData = (unsigned long) *irData;
    switch (irData[0]){
    case IR_ON_OFF:              // On/off control.  Invert
      ToggleSwitchState0;        // state of switch and
      break;                     // control LED.
    case IR_VOL_UP:              // Volume up control.
      ChangeSwitchLevel(2);      // Increase brightness.
      break:
    case IR_VOL_DN:              // Volume down control.
      ChangeSwitchLevel(-2);     // Decrease brightness.
      break;

    delay(12000);                // Ignore the other two outputs



// Quadrature dial input task-Read data from shaft encoder.
when(io_update_occurs(ioDial)) {
  ChangeSwitchLevel(input_value);


// Push button input task-Read data from on/off push button.
when(io_changes(ioButton) to 1){
  ToggleSwitchState();
  delay(500);                    // Debounce
}
```


Natural Voice Playback

Add a ***recorded natural*** voice to your system. Voice libraries of up to 255 words or phrases (2 min total max)-record your own using our optional **SDS-1000** development system and your IBM compatible, or we'll prerecord your messages for you. Eprom voice storage means your library is unaffected by power loss.

- Repeater identifiers
- Site alarms
- ANI
- Remote telemetry
- ATM's
- Multiple languages
- Emergency announcements

Several different models available

Palomar Telecom, Inc.
1201 Simpson Way • Escondido, CA • 92029
619-746-7996 • **FAX:** 619-746-1610

#201

X-10, LEVITON

Complete line *in stock*

TW523 kit (DOS)
$65.00

TW523 kit (Windows)
$90.00

6381 WI
$31.00

Baran-Harper Group Inc.
Voice: (905) 946-2451 • Fax: (905) 479-0455
BBS: (905) 479-0469

#202

#203

Since the technology complies with signaling regulations in North America and Europe, developers are able to expand their potential market significantly.

DEVELOPING AN INTEROPERABLE IR DIMMER

To give you an idea of how to take advantage of this technology, I will work through a simple example. You will see how NodeBuilder can be used to develop an interoperable, remote-controlled dimmer for the home.

The IR dimmer is a wall-mount dimmer controller with a quadrature dial and push button for manual input. An infrared receiver offers input from a hand-held remote controller. A single LED output is used as an on/off indicator.

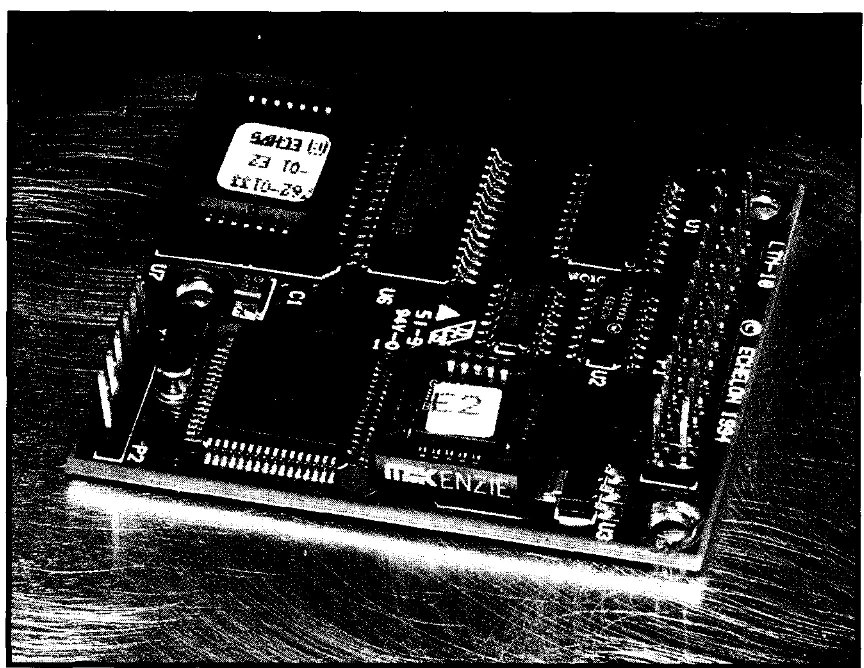


Photo 2: *The LTM-10 module is used for prototyping andproduction. It includes a Neuron 3150 chip, 32-KB flash memory, and 32-KB RAM.*

FIRST THE SOFTWARE

Applications for the Neuron chip are written in the Neuron C programming language. Neuron C is based on ANSI C, with extensions for network communications, I/O, and event-driven task management.

Network communications for interoperable LONWORKS devices are performed using LONMARK objects. These objects define standard formats and semantics for how information is exchanged between devices on a network. The most common objects are LONMARK sensors and LONMARK actuators. A sensor object corresponds to a physical device which can be monitored, whereas an actuator object corresponds to a physical device which can be controlled. For the IR dimmer, there is a single LONMARK sensor object.

Each object is defined by a unique object type number and a defined collection of network variables. To a Neuron C application, each network variable looks like a standard C variable. Unlike the standard C variable, however, network variables can be connected between devices. Therefore, updates to a network variable on one device automatically update the connected network variables on other devices,

Network variables have types like C variables, but a predefined set of Standard Network Variable Types (SNVTs; pronounced "snivits") go beyond C types by also defining standard units and ranges. For example, SNVTs are defined for temperature, pressure, and velocity.

Another difference from standard C variables is that network variables have a direction. Output network variables automatically send their values to other devices when updated. Input network variables are automatically updated when they receive updates from other devices.

For the IR dimmer, there are two output network variables: nvoSwitch and nvoRawHwData. The nvoSwitch output reports the on/off state and

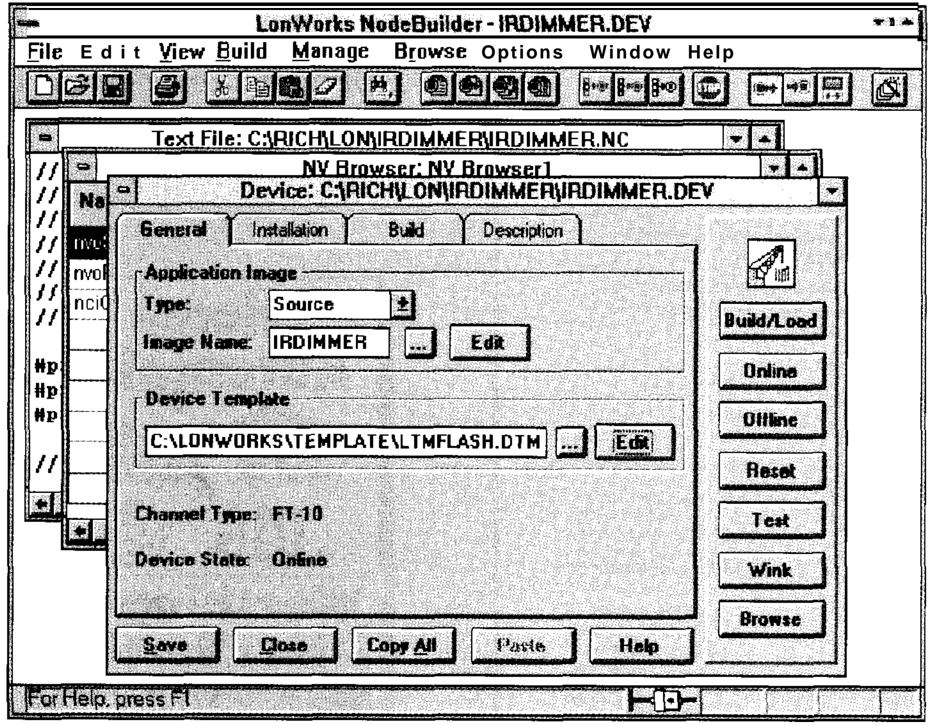


Photo 3: *This device definition specifies the application code and hardware device template to be used for the IR dimmer.*

level of the dimmer. This output can be connected to network lamp modules which control their level. However, the output can be connected to other devices as well. For example, by connecting a networked amplifier device, you could control the on/off state and volume of the amplifier output.

The `nvoRawHwData` output reports valid infrared commands and could implement other types of IR control. You could connect this output to a central controller to invoke control applications for the home network.

Listing 1 contains Neuron C statements which specify that the IR dimmer has a single LONMARK sensor (object type 1) and declares the two network variable outputs.

Once declared, output network variables are updated with a simple C assignment statement. For example, the following C statement assigns a new value to the IR dimmer sensor output value:

```
nvoSwitch.state =!nvoSwitch.state;
```

Interfacing to I/O devices is as simple as network variables. Table 1 lists the 33 built-in device types that Neuron C includes. These types provide built-in support for the most commonly used I/O devices in home control.

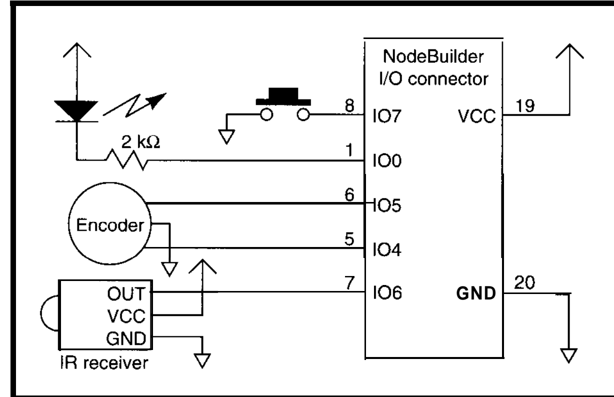


figure 1: *The schematic for the IR-dimmer prototype shows how most of the I/O interface is implemented internally in the Neuron chip.*

Interfacing to any of these types is done by declaring an I/O object and then reading or writing it with a function call. For example, Listing 2 declares the five I/O objects for the IR dimmer.

The following statement reads the IR sensor:

```
io_in(ioIRData, irData, 12,
  65424UL, 65424UL + 59UL)
```

The input parameters to the `io_in()` call define the number of bits (i.e., 12) per command and the threshold period to distinguish between the one and zero input. These parameters are selected for a Sony RM-V10 remote control. Other remote controls can be used by changing the timing parameters. For this project, all testing was done with a Sony remote control and a Sony-compatible universal remote control.

Processing for network variables and I/O objects is accomplished within tasks. Neuron C tasks are independently scheduled statement sequences. Each task is defined by one or more when statements that specify the

HCS II
Home Control System

Energy Management
Security and Alarm
Coordinated Home Theater
Coordinated Lighting
Monitoring and Data Collection

Get all these capabilities and more with the Circuit Cellar HCS II. Call, write, or FAX us for a brochure. Available assembled or as a kit.

Circuit Cellar HCS

CIRCUIT CELLAR, INC.
4 Park Street, Suite 12 • Vernon, CT 06066
Tel: (203) 875-2751 • Fax: (203) 872-2204

#204

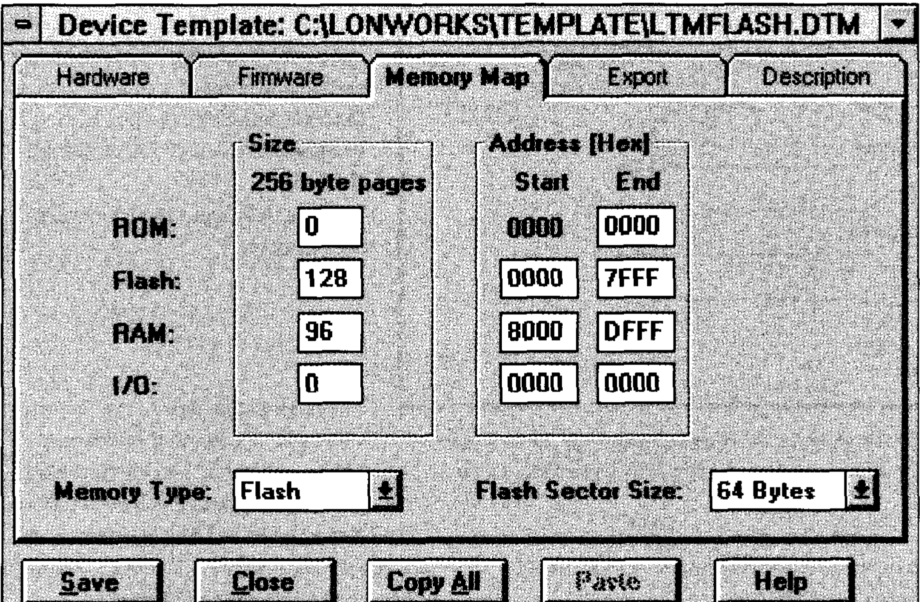


Photo 4: *This device template defines the hardware configuration for the IR dimmer.*

events that must be true before the task can be scheduled.

The complete code for the IR dimmer is shown in Listing 3. There are three tasks: **one** that executes when an IR input is received, one that executes when the quadrature input dial is moved, and one that **executes when** the on/off button is pressed.

DESIGNING THE HARDWARE

LonWorks applications can be designed around any of the three Neuron 3120*xx* chips (for description of the chips, see the Neuron Chip sidebar), the Neuron 3 150 chip with 2 KB of on-chip RAM and up to 58 KB of external memory, or any other microcontroller as long as the Neuron chip is used as a communications coprocessor. Because of the extensive support provided by the Neuron chip firmware, the IR dimmer application requires only 486 bytes of memory and easily fits in a Neuron 3 120 chip.

The IR dimmer was prototyped using the NodeBuilder hardware. The LTM-10 node included with Node-Builder provides a complete prototype node. The infrared decoder, quadrature dial, input button, and output LED were constructed on a prototyping board **shown** in Photo 1. This board was plugged into the Node-Builder hardware. The schematic for the prototype I/O board is shown in Figure 1.

Prototypes may also be easily constructed using the LTM- 10 module (see Photo 2). The LTM-10 module includes a Neuron 3 1.50, a 10-MHz

The Neuron Chip

The Neuron chip (see Photo I) uses advanced CMOS VLSI technology to implement low-cost control networks. Included in each Neuron chip are all the functions required to acquire and process information, make decisions, generate outputs, and propagate control information via a standard protocol. Communication takes place across a wide variety of network media such as twisted-pair cable, power line, infrared, radio frequency, or coaxial cable.

Neuron chips are manufactured and distributed by Motorola and Toshiba. They are available in four versions: the 3120, 3120E1, 3120E2, and 3150 chips. All versions are highly integrated, require a minimal number of external components, and include three 8-bit CPUs.

One CPU executes user applications, which could include measuring input parameters, timing events, making logical decisions. and driving outputs.

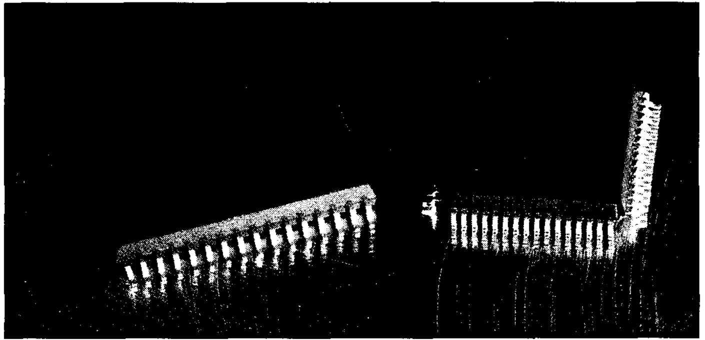


Photo I: *The Neuron 3120 and 3150 chips are manufactured and distributed by Motorola and Toshiba.*

The second CPU executes the LonTalk protocol. Messages are properly encoded and decoded for distribution over the network. This protocol supports distributed, peer-to-peer communication that enables individual nodes, such as actuators and sensors, to communicate directly with one another.

The third CPU controls the Network Communication Port, which physically receives and the packets. There is onboard EEPROM and RAM, and either onboard ROM (Neuron 3 120*xx* chip) or an external memory port (Neuron 3 150 chip) to support the three CPUs.

Table I summarizes the memory configurations of the four Neuron chips.

	Neuron Chip			
	3120	3120E1	3120E2	3150
EEPROM bytes	512	1024	2048	512
RAM bytes	1024	1024	2048	2048
ROM bytes	10.240	10.240	10.240	0
Ext. Memory Interface	No	No	No	Up to 58 KB

Table I: *Neuron chip memory configurations provide a range of options for memory size and integration.*

NV Browser: NV Browser1

Name	Format	Poll	Dir	Interval	Len	Value
[illegible]	[illegible]	On	Out	2	2	[illegible]
nvoRawHwData	SNVT_count	On	Out	2	2	0
nciGain	SNVT_count	On	In	20	2	5

Photo 5: *The network variable browser makes it easy to observe and manipulate the IR dimmer over the network.*

crystal, 32-KB flash memory, and 32-KB RAM. The I/O and communications pins are all 0. 1″ centers for easy prototyping.

DEFINING THE DEVICE

A device in NodeBuilder is defined using a device file. The device file defines the device's hardware characteristics and specifies which Neuron C application the device needs. The screen shot in Photo 3 shows the device definition for the IR dimmer.

The IR dimmer device is defined by specifying the application program tobethe **IRDIMMER.NC**file described earlier. For prototyping, the device template is defined to as **LTMFLASH** to specify that the hardware will be based on the LTM- 10 LONTALK module with the application stored in the LTM-10 flash memory.

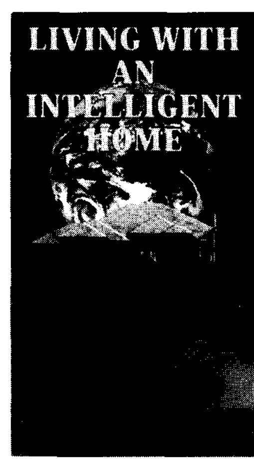
When the device is ready for high-volume production, the device template can be changed to the 3 120 template. The default device templates simplify hardware definition, but a custom template can be defined for any hardware configuration. The device template is easily modified by clicking on the Edit button next to the template name. Photo 4 shows the memory tab of the device template for the LTM-IO module.

PROGRAMMING THE DEVICE

With the Neuron C application written and the IR dimmer device defined, you are ready to compile the program and program the device. You do this by simply clicking the Build/Load button in the Device window shown in Photo 3. This automatically installs the device hardware, invokes the compiler and linker with the parameters specified in the device file, downloads the new application to the device, and starts the new

Let's Work Together.

Networking provides access to a world of resources, and Home Systems Network offers a world of resources to those who are interested in home automation. *Check it out.*



◆ **Are you looking for information?**
Obtain unbiased information about how to install and use all types of home automation systems from our books and *Intelligent Home* video tape series.

◆ **Are you looking to identify sources?**
Call our toll free number for a list of sources for any type of home automation dealers, products, or services.

◆ **Are you looking for marketing assistance?**
List your products and services in the Home Systems Network database and let us tell the world about them through our books, video tapes, television shows and referral services.

HOME SYSTEMS NETWORK **P.O.** BOX 3006 EDMOND, OK 73083 (800) **808-0718**

#205

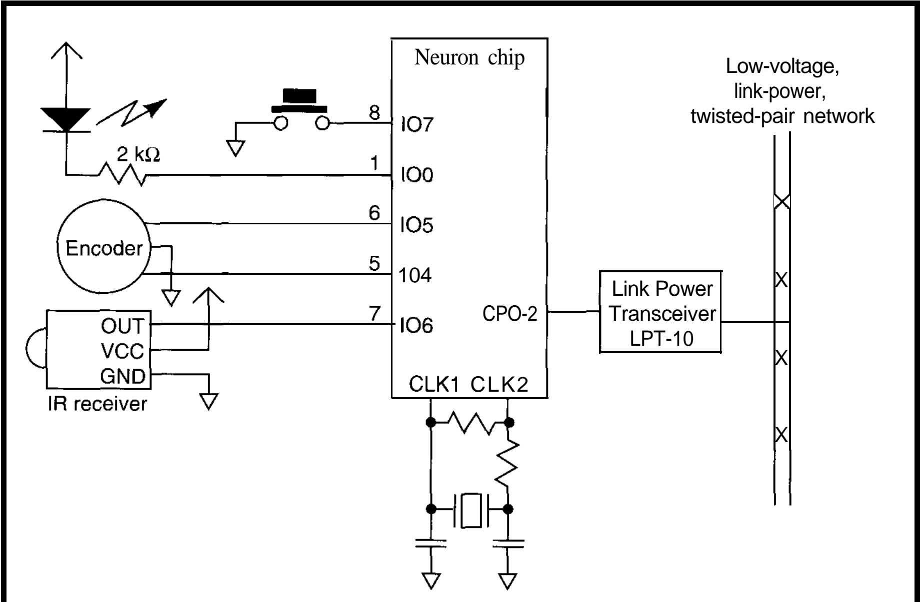


Figure 2: *The IR dimmer device with a link-power twisted-pair transceiver provides the simplest implementation since no local power supply is required. The LPT-10 transceiver supplies sufficient power for the entire IR dimmer device. Other transceivers can be used in place of the LPT-10 to communicate on other media without having to change the core of the design*

application. The downloading occurs over the network during development.

Again, when a device is ready for production, the programming can be done using a Neuron 3 120 programmer for Neuron 3 120*xx*-based devices or using a standard PROM programmer for Neuron 3150-based devices.

TESTING THE DEVICE

The IR dimmer device is tested over the network, exercising it using the same interface that will be used by other LonWorks devices when it is installed in the network. Clicking on the Browse button in the Device Window opens the Network Variable Browser window shown in Photo 5.

By default, all the network variables on the device are displayed in the left column, followed by the type, size, and current value of the network variables. The browser automatically polls all the network variables on the device and updates their values.

The operation of the IR dimmer device is tested by sending infrared commands, rotating the quadrature dial, pressing the push button, and observing the resulting network variable changes. If the network variables change as expected, the application is working and ready to go to production.

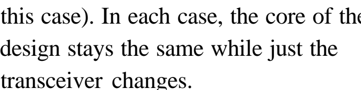
If developers are not sure about the remote controller command numbers, they can observe the `nvoRawHwData` output network variable and determine their values. If the application doesn't work as expected, the developer modifies the Neuron C application, reruns Build/Load, and tests again.

A source-level debugger ships in summer '9.5 as a free upgrade for all NodeBuilder 1.0 customers. Until then, the network variable browser can be used for debugging and testing LonWorks devices.

PRODUCING THE DEVICE

Once the design is verified with the prototype hardware, a production version of the hardware can be built using control modules for quicker time to market or using a full custom design.

Figures 2 shows a complete custom design for the IR dimmer. It uses an LPT-10 link-power twisted-pair transceiver for a hard-wired implementation with link power. The transceiver supplies all the power required by the device, so a separate power supply is not required. Another alternative is to use an FTT- 10 free-topology twisted-pair transceiver (in place of the LPT-IO) for an isolated twisted-pair design requiring local power. A third alternative is to use a PLT-20 power-line transceiver for easy installation into the home (a separate power supply is required in


this case). In each case, the core of the design stays the same while just the transceiver changes.

INSTALLING LonWorks DEVICES

Typically, one node of a LonWorks network installs all the other nodes on the network. This installation tool can be integrated into a home computer or set-top box connected to the network. Developers can also build this tool themselves or use an existing tool for home networks such as Windows-based tools from IBM in Germany or Control Plus in the U.S.

CONCLUSION

With the availability of Node-Builder ($3995 at the time of this writing), every device developer in the home automation market can start building LonWorks-based products. The availability of low-cost Neuron chips, OEM modules, and software makes the development of easy-to-install, reliable, and low-cost LonWorks devices a reality.

Rich Blomseth is Echelon's product marketing manager for development and network services products. He has been involved with the design and development of control networks since 1978, and has been at Echelon since 1989. Rich has an M.S. in Computer Science from the University of California, Berkeley. He may be reached at richardb@netcom.com.

SOURCES

Echelon Corporation
40 15 Miranda Ave.
Palo Alto, CA 94304
(415) 855-7400
LonLink BBS: (415) 856-7538
telnet://lonlink.echelon.com
ftp://lonworks.echelon.com

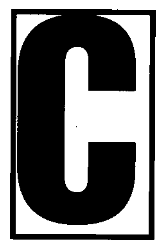
I R S

413 Very Useful
414 Moderately Useful
415 Not Useful

CEBus for the Masses

PETER HOUSE

Picking up where other CEBus articles in INK left off, Peter introduces us to packet construction and CAL messages. By the end, you'll be able to control a real-live CEBus light switch!

EBus is the Electronic Industries Association's (EIA) open standard IS-60 describing a method of communication between electronic products in the home using five different media: power line, twisted pair, coax, broadcast RF, and infrared. A sixth medium, fiber optic, has a section left open and is undefined at this time.

CEBus is a complete, packet-oriented, peer-to-peer network using a Carrier Sense Multiple Access with Collision Detection and Collision Resolution (CSMA/CDCR) protocol. The CEBus standard defines everything needed up to and including the language used for application-to-application communication called the CEBus Common Application Language (CAL).

In this article, I'll introduce you to packet construction and show you how to create CAL messages that control a CEBus light switch. Hang on-or the details may swamp you!

CEBUS AND CAL

The CEBus protocol is described using the OSI/ISO seven-layer model. CEBus uses four of those layers: application, network, data link, and physical. Note the actual application function (e.g., a light switch) is distinct from application layer protocol.

WHY CAL?

The CEBus application language is a set of common language and data constructs created to enable interoperability between products used in residential automation. This interoperability is available between different manufacturers' products even without prior knowledge of the products.

For example, information to control Company X's light module or stereo is published by the EIA or the CEBus Industry Council (CIC). This information is known to the world without having to know anything specific regarding Company X's design implementation of how they use a class A amplifier to control a vacuum-encased, electrothermal photon emitter-otherwise known as a light bulb.

PACKET STRUCTURE

A CEBus packet frame can be broken down into several parts: the Link Protocol Data Unit (LPDU), the Network Protocol Data Unit (NPDU), the Application Protocol Data Unit (APDU), and the CAL message. I describe these different parts using a mailed letter (see Figure 1) as an example. Figure 2 shows a breakdown of a packet structure illustrating the different parts.

LPDUHEADER

The LPDU header contains the control field and the source and destination addresses. In the letter mailing scenario, the control field represents the postal service used to send the letter. The control field specifies the packet type,

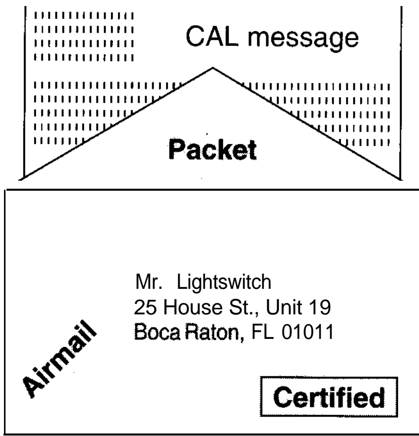


Figure 1: *A letter illustrates the various services availabe to CEBus packets.*

packet priority, and service class to the Data Link Layer (DLL). Figure 3 shows a bit-oriented breakdown of the control field.

The packet type is used to select the form of DLL service. This method roughly corresponds to sending a letter normal mail or with a return receipt requested. The DLL handles all channel acquisition, timing, and packet-receipt verification.

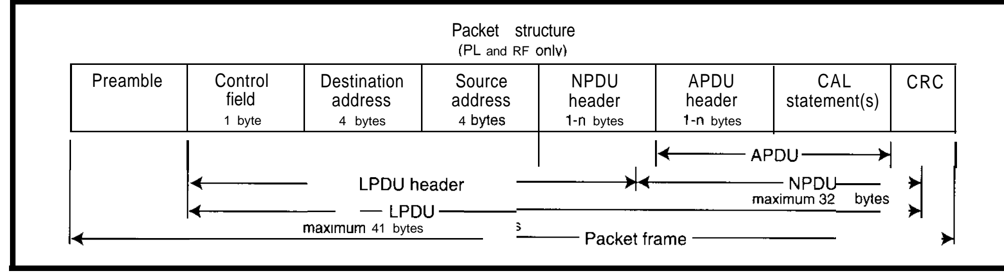


Figure 2: *The elements of a CEBus packet are broken down into logical groups with size information*

There are two classes of DLL service: acknowledged and unacknowledged. Acknowledged services expect a response from the receiving nodes DLL and unacknowledged services do not. DLL packet types include immediate acknowledge (IACK), acknowledged data (ACK_DATA), unacknowledged data (UNACK_DATA), failure (FAILURE), addressed acknowledged data (ADDR_ACK_DATA), addressed immediate acknowledge (ADDR_IACK), and addressed unacknowledged data (ADDR_UNACK_DATA).

Once a node acquires the channel, the response from the receiving node is considered part of the acquisition. The acknowledge packet must start within 200 µs after the end of receiving a packet from the requesting node. After the DLL receives the transmit request from the application, the DLL automatically handles all of the retries and channel acquisition.

The ACK_DATA services' acknowledge is an ultrashort packet with only an NPDU header and a null information field. The acknowledge packet's control field contains either FAILURE or IACK packet type. The destination and source address fields must be null. IACK signifies to the transmitting DLL the proper receipt of the packet. FAILURE signifies that the receiving node's DLL is operational but could not pass the packet to its network layer.

The source address is optional in the ACK_DATA packet and can be omitted to reduce channel-access duration. If the transmitting node's DLL does not receive an IACK, a retry must begin within 600 µs. If the retry does not receive an IACK, the DLL passes an error back up the protocol stack.

ADDR_ACK_DATA service supports additional capabilities and enhances reliability. A one-bit sequence number is used by the receiving node to ignore duplicate packets from the transmitting node during a predetermined time interval defined in the CEBus specification. Because of this added feature, the DLL accesses the channel multiple times to make sure a packet using ADDR_ACK_DATA service is transmitted.

With the ADDR_ACK_DATA, the receipt packet includes the ADDR_IACK type in the control field. As well, the destination addresses must be present—which means that the source and destination address must also be present in the requesting ADDR_ACK_DATA packet.

If the transmitting node's DLL does not receive an IACK, a retry must begin within 600 µs. If the retry does not receive an IACK, the DLL relinquishes the channel. It may reaccess the channel and attempt to repeat the transmit process without passing an error up the stack. Only if the DLL exhausts all of the predetermined channel-access attempts is an error reported.

ADDR_UNACK_DATA has similar capabilities to ADDR_ACK_DATA service without acknowledgment packets or immediate retries. For ADDR_UNACK_DATA, the DLL transmits multiple copies of the packet using multiple channel accesses.

Packets using a broadcast address must use unacknowledged services (either UNACK_DATA or ADDR_UNACK_DATA) since the acknowledgments from the many receiving nodes would collide and result in unreceivable noise. ADDR_UNACK_DATA is the preferred service for broadcast packets since multiple packets using multiple channel accesses are possible and result in higher reliability.

Packet priority is used by the DLL to determine the channel-access priority timing. To gain access to the channel, a node first listens for channel activity (carrier sense). If there is activity, the node waits until it is finished. After a fixed amount of time (based on priority) plus a random amount of time, the node can attempt to gain channel access by sending a random-number packet preamble used for contention resolution. If no contention is detected, the packet is sent. If contention is detected, the node must wait for a new channel access and transmission attempt.

The earliest a packet with the highest priority can start is 1 ms after the previous packet ends. The only

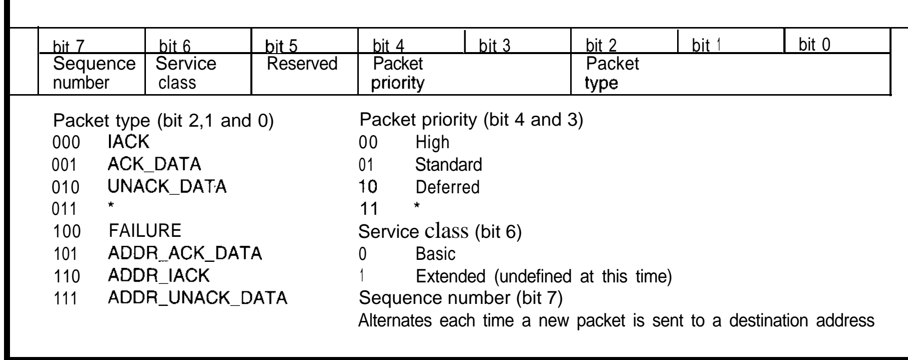


Figure 3: *The LPDU header sets the Data Link Layer services and chooses the media access priority.*

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Privilege	Routing		Packet flag	Extended services	Allowed media	Brouter subfield	

Privilege
0 Unprivileged
1 Privileged
Routing
00 Request_ID
0 1 ID_packet
10 Directory route
11 Flood_route
Packet flag
0 First packet
1 Only packet

Extended services
0 No extended services
1 Extended services octet to follow
Allowed media
0 This media only
1 Allowed media octet to follow
Brouter
00 No brouter address
01 First brouter address presence
10 Second brouter address presence
11 First and Second brouter address presence

Figure 4: *The NPDU header describes network information including allowed media and how the packet is routed.*

two exceptions to this are for a packet retry and acknowledgment. An acknowledgment must start within 200 µs after the end of the packet to be acknowledged. A packet retry is sent approximately 600 µs after the previous packet ends.

The sequence number is a single bit field and is alternated for each packet sent to a destination address. This enables the DLL to distinguish a received packet which is a copy and not pass it up the stack to the application layer. A packet could be a copy due to a transmitting node using ADDR_UNACK_DATA with duplicate packets or using ADDR_ ACK_DATA, in which a sending node does not hear the acknowledgment and sends a retry.

DESTINATION AND SOURCE ADDRESS

The destination address is four bytes long giving CEBus a potential of four giganodes. The address is divided equally into two logical portions: system address and Media Access Control (MAC) address—usually called the house and unit codes since most people are already familiar with these terms.

If the unit code is zero, it is considered a house broadcast address-all nodes respond regardless of their unit code. If the house and unit codes are both zero, then all nodes respond because this is considered a global address. The destination address has the same formatting as the source address and is transmitted in the same order.

The address is placed in the packet from the unit code's least-significant bit of the least-significant byte to the house code's most-significant bit of the most-significant byte. This seemingly reverse ordering offers protection.

For instance, when the bits are actually transmitted over the channel, the DLL suppresses leading zeros to reduce the transmitted time of the packet and improve network throughput. Suppression of leading zeros is possible because end-of-field separator tokens are inserted by the DLL before the packet is transmitted.

NPDUHEADER

The NPDU header specifies how the packet is sent. Using the mail analogy, it corresponds to using air mail, normal delivery, or "in care of" when a router transfers a packet from one medium to another (e.g., twisted pair to power line). The NPDU header consists of six fields: privilege, routing, packet flag, extended services, allowed media, and brouter. Figure 4 shows a bit-oriented breakdown of the NPDU header.

The privilege field is restricted to packets related to system management.

The routing field sends an ID packet, request for the recipient to send an ID packet, and selects directory or flood routing from a router. An ID packet is sent out by a configured device whenever it is powered on as a sign-on message or when requested by a router. A router uses the ID packet to keep a list of nodes for each supported medium.

The packet flag field specifies if this is the only packet or the first packet of a multipacket message. Long messages can be segmented into several packets since the maximum packet length is 41 bytes, with nine used for control and addressing. This leaves 32 bytes for the complete NPDU including any CAL statements.

The extended services field specifies that additional NPDU bytes follow with additional NPDU services.

The allowed media field tells routers and brouters if they should route the message to another medium. If you had a PL-to-IR brouter, you probably would not want to route the messages to IR because IR is typically used for hand-held remotes or portable devices. If the allowed media field specifies other media, an additional NPDU byte specifies the allowed media.

The brouter field is used to control routing of packets originating or terminating on wireless media. A brouter is a device used to cross between wireless and wired media. For instance, you may want a hand-held IR

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Reserved	Mode	Type			Invoke ID		

Reserved
1 Must be 1
Mode
0 BV-Basic variable length
1 BF-Basic one byte fixed
Type
000 Not used
001 Reject
010 Result
011 Receipt acknowledge
100 Implicit invoke
101 Explicit invoke
110 Conditional invoke
111 Explicit retry

Invoke ID
000 A three bit increment identifier used for packet tracking
001
010
011
100
101
110
111

Figure 5: *The APDU header specifies how the receiving application layer should respond to the packet.*

remote control to send commands. Your TV could act as a brouter to the VCR and stereo via the power line.

	Explicit Invoke	Implicit Invoke	Conditional Invoke
SetValue	response	no response	no response
GetValue	response	no response (invalid)	response

Table 1: *Whether a response is generated for the SetValue and GetValue methods depends on how they were invoked.*

APDU HEADER

The APDU header specifies how and if the receiving application layer should respond to the packet. There are three fields in the APDU: mode, type, and invoke ID. In our mail analogy, the APDU is like an RSVP at the bottom of the CAL message letter. It tells the CAL interpreter on the receiving end if it should respond (also called *end-to-end confirmation)* or if there are enclosures or a follow-up letter. Figure 5 shows a bit-oriented breakdown of the APDU header.

The mode field tells you whether the APDU uses multiple bytes. Most messages use the basic fixed-length APDU mode. Additional bytes are used for services such as authentication or encryption.

Authentication is used by the receiving node to verify the sending node's authority before the application layer passes the rest of the APDU to the CAL interpreter. Encryption sends packets with the message secured.

The type field has seven values: reject, result, receipt acknowledge, implicit invoke, explicit invoke, conditional invoke, and explicit retry. Reject is sent from the receiving node's application layer which rejects the packet for some reason. Result and receipt acknowledge are sent from the receiving node's CAL interpreter to tell the sending node's CAL interpreter that the CAL command has been completed or initiated with a complete response to follow.

Implicit invoke tells the receiving node an application level response is not necessary. Explicit invoke tells the receiving node to respond with a CAL result response. Explicit retry expects acknowledgment from the receiving node's application layer within a predetermined amount of time-acknowledgment could be either result or receipt acknowledge. If none is received, the application layer (not the application) automatically retries the message.

Conditional invoke enables a device to send a response only if it has a nonempty result to return. If there is a result to return, the response packet contains a result type in the APDU header type field. Conditional invoke could be used with a broadcast address. A response result would only be initiated by a node matching the conditional criteria since there would be a result only if the condition was true.

A SetValue CAL method does not normally have a response. If the transmitting node wants to make sure the SetValue method was handled properly by the receiving application's CAL interpreter, an explicit invoke can be used. Table 1 demonstrates the differences between the invokes and their set and get values.

Invoke ID is a 3-bit field incremented (and rolled over) for each new transmitted message to a destination address. The application responds with a reply packet using the same value in the invoke ID field. Invoke ID, along with the destination address, is used by the sending node so when the results come back, the sending node matches the result application packet to the proper command.

A transmitting node cannot stack or send more than one command to a receiving node until the receiving node responds to the first packet. A transmitting node sends packets to multiple destinations and uses invoke ID and the destination address to sort out the result responses.

Let me take a moment to clarify the distinction between the LPDU packet type and the APDU type. The DLL acknowledgment, if requested in the LPDU packet type field, takes place regardless of the APDU type and without the application's knowledge if the application requests acknowledged service.

As far as the application layer is concerned, it is communicating with the application layer of the receiving node. The application layer is unaware of any retries at the DLL layer and the application is unaware of any retries by the application layer.

Figure 6 shows two nodes-one node is sending an implicit invoke in the top example and the other is sending an explicit invoke to the bottom example. In the top example, the application requests ACK_DATA DLL service, and in the bottom example, UNACK_DATA DLL service is requested.

The application passes the packet to the DLL. If the DLL cannot get an IACK, only then does the DLL notify the application of an error. When the application layer calls for a response from the receiving-node application layer using the explicit invoke APDU type, the receiving node returns a result response packet to the original sending node. This activity is separate

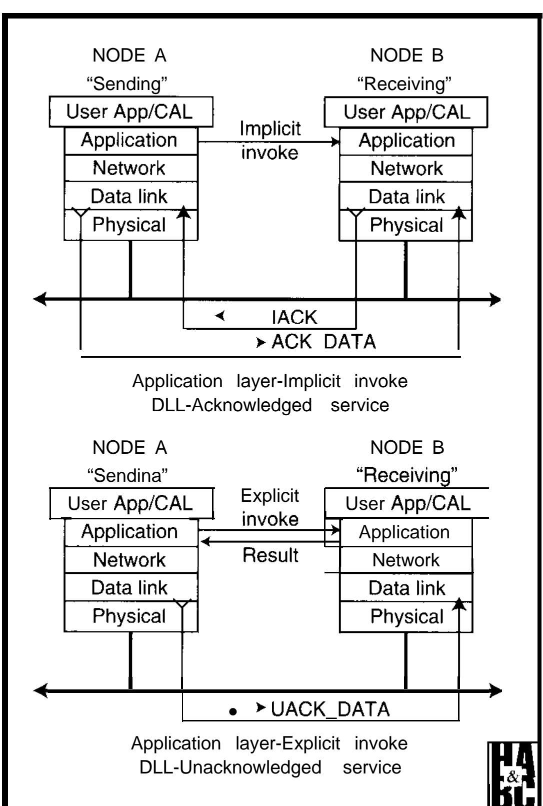


Figure 6: *Here's a breakdown of the OSI layers showing the difference between CEBus Application Layer and CEBus Data Link Layer services.*

from the lower level DLL acknowledgment services and can be used regardless of the DLL service.

CAL DEVICE MODEL

CEBus uses a hierarchical model to describe each node. Each node includes two or more contexts, each made up of two or more objects. Each object contains one or more Instance Variables (IVs).

UNIVERSAL CONTEXT

The first context in every node is called the *universal context* and has nothing to do with normal operation of the actual device. The universal context is numbered 00 and controls the device's presence on the CEBus network.

The universal context consists of two objects: the node-control object (object 0) and the context-control object (object I). The **node control object** has IVs to hold universal device information such as the device addresses, manufacturer name, and other device management information, while the context control object has a single IV called *object_list*, which holds a list of object IDs for this context. Every context contains one or more IVs, which control or publish some aspect of the device. The universal context is required in every CEBus-compatible product.

Value	Name
01	Node Control
02	Context Control
03	Data Channel Receiver
04	Data Channel Transmitter
05	Binary Switch
06	Binary Sensor
07	Analog Control
08	Analog Sensor
09	Multiposition Switch
OA	Multistate Sensor
0B	Matrix Switch
0C	Multiplane Switch
0D	Ganged Analog Control
OF	Meter
10	Display
11	Medium Transport
13	Dialer
14	Keypad
15	List Memory
16	Data Memory
17	Motor
19	Synthesizer/Tuner
1A	Tone Generator
1C	Counter
1D	Clock

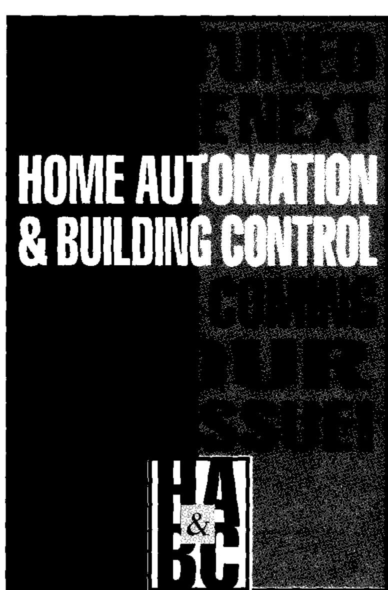
Table 2: The CAL objects published by the CEBus Industry Council are combined together to model any real-world device.

Value	Mnemonic	Basic Syntax	Data Types
40	nop		
41	**setOFF**	IV	B
42	**setON**	IV	B
43	**getValue**	IV	BNC
44	**getArray**	IV [, [<offset>], <count>]	D
45	**setValue**	IV	BNC
46	**setArray**	IV [, [<offset>], <data>]	D
47	add	IV1, IV2, [IV3]	N
48	increment	IV [,<number>]	N
49	subtract	IV1, IV2, [IV3]	N
4A	decrement	IV [, <number>]	N
40	compare	IV1, IV2	BNCD
4C	comparei	**IV1**, <data>	BNCD
4D	copyValue	**IV1**, IV2 [, <context>, <object>]	BNCD
4E	swap	IV1, IV2	BNCD
52	exit	[<error number>]	
53	alias	<alias ID> [<string>]	
54	**inherit**	IV, <value>	D
55	**disinherit**	IV, <value>	D
56	if	<boolean> BEGIN <msg list> [else clause] END	
57	do	<boolean> BEGIN <msg list> END	
58	while	<boolean> BEGIN <msg list> END	
59	repeat	<boolean> BEGIN <msg list> END	
5A	build	<macro ID> BEGIN <message list> END	

Methods in bold are required for minimum CEBus implementation; "," is F5 delimiter

Table 3: *CAL methods perform operations on CAL instance variables.*

#209



#210

OTHER CONTEXTS

CEBus defines contexts which can be grouped together to define just about every device imaginable. For instance, the lighting-control context includes parameters for defining a light switch. For a more complex device like a stereo receiver or a TV, several contexts containing various objects can be combined. For this article, I will focus on a light switch available in a 500-W dimmer version or a 15-A relay version. Refer to the CAL model for the light switch shown in Figure 6 and the CAL object list in Table 2.

The lighting context has two objects. The context control object is required to be the first object in every context with the exception of the universal context. The context control object has one IV called object_list, which holds a list of the objects in this context. Here, it would be 02 01 07 02, showing this context has a CEBus object type of 02 for the first (01) object and an object type of 07 for the second (02) object. The current_value IV is required in the light-control object to be CEBus compatible.

The CAL object 07 is an analog-control object and has 14 IVs as published by the EIA. A manufacturer can choose to implement only those IVs which make sense for a particular product or add nonstandard IVs. Unfortunately, there is no way for anyone to know what nonstandard IVs are if they wanted to use them since the IVs are not readily available until after the manufacturer chooses to publish them.

Only five IVs are implemented in this light switch. The current_value stores the current dim value in percent (o-100). The saved_value temporarily saves the current_value. The step-rate and step_size set the ramp rate of the current_value IV used for dimming and feature_select manipulates the current_value IV and controls the light.

Name	Value	Name	Value
DO	57	DELTA	ED
WHILE	58	PARAMETER	EE
REPEAT	59	NULL	FO (reserved)
BUILD	5A	MINIMUM	F1 (reserved)
AND	E0	MAXIMUM	F2 (reserved)
OR	EI	DEFAULT	F3 (reserved)
NOT	E2	DATA	F4
XOR	E3	DELIMITER	F5
GT	E4	ESCAPE	F6
GTE	E5	BEGIN	F7
LT	E6	END	F8
LTE	E7	END_OF_CMD	F9
EQ	E8	END_OF_LIST	F A
NEQ	E9	END_OF_MSG	F B
ELSIF	EA	END-OF-FILE	FC (reserved)
ELSE	EB	Error	FD
LITERAL	EC	Completed	FE

Table 4: *The CAL tokens are unique symbols in the CAL message, which are used as delimiters and to create programming constructs.*

METHODS, TOKENS, AND ERROR CODES

CAL methods are used to perform operations on CAL instance variables. SetValue and GetValue are probably the most used and are shown in the example packets later in the article. Table 3 shows a list of the CAL methods.

CAL tokens are used to create programming constructs and for delimiting data. The Data (F4), Delimiter (F5), and Literal (EC) tokens are the most common tokens found in CAL messages. The Data token is used as a starting delimiter to separate array data from the preceding information. The Delimiter token separates portions of a CAL message. The Literal token precedes string data. Table 4 shows a list of the CAL tokens and their hexadecimal values.

Table 5 lists the error codes returned from a CAL interpreter. These error codes appear following an APDU with a type equal to reject. In a multiple-part command, each part has a corresponding APDU header and error code.

Value	Description
0	Unknown Context ID
1	Unknown Object ID
2	Unknown Method ID
3	Unknown IV Label
4	Malformed Expression
5	Macro not defined
6	Alias not defined
7	Syntax error
8	Resource in use
9	Command too Complex
10	Inherit Disabled
11	Value out of Range
12	Bad Argument Type
13	Power Off
14	Invalid Argument
15	IV Read Only
16	No Default
17	Cannot Inherit Resource
18	Precondition Complete
19	Application Busy

Table 5: *CAL error codes are used to indicate various application-layer error conditions.*

DATA TYPES

There are four data types used in CAL: string, data, numeric, and Boolean. Strings are delimited by CAL tokens or are at the end of a packet. Data can be thought of as array oriented. Numeric is usually represented by a string of ASCII numbers (e.g., 31h 30h 30h for the number 100). Boolean is always true or false. The byte Olh is true and 00h is false.

PACKET BUILDING

There are many things you can do to a CEBus light switch by sending it packets. In this example, we turn it on and off, set a dim level, ramp to a level, read the serial number, and change the device address based on the serial number.

In the LPDU, we set the packet priority to STANDARD and the packet type to ADDR_UNACK_DATA. All other fields are zero for a control byte of 0Fh. Remember the sequence number is set by the DLL—we don't actually have control of it.

For this demonstration, we assume the light switch has a house code of 5 and a unit code of 1. The controller (us) has a house code of 2 and a unit code of 1.

The NPDU byte has a value of 50h. This value calls for unprivileged, directory-routed service on this medium only.

The APDU is a single byte with a value between E8h and EFh. This specifies a mode of basic one-byte fixed and a type of explicit invoke. The invoke ID increments for each packet sent.

CAL MESSAGES

In the following examples, the first 11 bytes of each packet are the same with the exception of byte 11, where the invoke ID field is incremented. The first 11 bytes (hex) are:

0F 01 00 05 00 01 00 02 00 50 E0.

As you recall, the first byte is the control byte and the next four bytes

are the destination-address information in right-to-left order, followed by the source address, also in right-to-left order. The next two bytes are the NPDU and APDU headers.

ON, OFF, OR DIM

The on command uses the SetValue method to send a value of 37h to the feature-select IV. This sets the current value to 100 by the definition of feature-select. The bytes (hex) then are:

21 02 45 66 F5 37.

The 21 is the ID of the lighting context, 02 is the object number of the analog control, 45 is the SetValue method, 66 is the feature_select IV (i.e., ASCII "f"), F5 is a delimiter, and 37 is an ASCII "7". The complete packet is:

OF01 00050001 00 02 00 50 E8 21 0245 66 F5 37.

The response to this packet is:

0F 01 00 02 00 01 00 05 00 50 D0 FE.

The LPDU (50) shows a standard packet priority and a packet type of ADDR_UNACK_DATA, which does not require a response from the DLL to acknowledge packet receipt. The source and destination addresses are reversed since the device is now sending to the controller instead of receiving from it. The NPDU (EO) is the same as transmitted. The APDU has the same mode, but the type field shows that it is a result packet with the result of FE, which is the completed CAL token.

Whew! I think I'll have someone get up and turn the switch on next time.

The packet to turn the light off is identical, except the value for the feature_select IV changes to 33h and the invoke ID field for the APDU is incremented by one. Setting the feature_select to 33h also saves the current_value in the saved_value IV before setting the current_value to 0. This offers the feature of having the light later restore to the previous dim setting.

The result packet is the same, except the invoke ID matches the invoke ID we sent, which is what the invoke ID is intended for. We could issue several commands to this light switch. Since the result packets are all identical, except for the invoke ID, we can use this field to track the responses to the packets sent. The complete sent packet is:

0F 01 00 05 00 01 00 02 00 50 E9 21 02 45 66 F5 33

and the response is:

OF01 00020001 00 05 00 50 D1 FE.

To dim the light, we set the current_value IV to the dim level desired. In this example, we'll use 50%. With a packet of 21 02 45 43 F5 35 30, the 21 is the ID of the lighting context, 02 is the object number of the analog control. 45 is the SetValue method, 43 is the current-value IV (ASCII "C"), F5 is a delimiter, and 35 30 is ASCII for "5" and "0" or 50%. The complete packet is:

0F 01 00 01 00 14 00 15 00 50 EA 21 02 45 43 F5 35 30.

The result packet is once again identical, except for the invoke ID:

OF01 00020001 00 05 00 50 D2 FE.

#211

GETTING THE SERIAL NUMBER

The serial_# **instance variable** "s" can be found in the node control object of the universal context in object 1. The packet reading the serial_# **has the same** first 11 bytes as above and the additional CAL command 00 01 43 73. The CAL command is in the universal context (00), object one (01), get the value (43) of instance variable "s" (73). The complete send packet is:

0F 01 00 02 00 14 00 15 00 70 EB 00 0143 73.

The result packet is:

8F 01 00 02 00 01 00 05 00 50 D3 FE EC 54 30 30 30 30 30 303030303539.

Note the incremented invoke ID in the sent packet and the matching invoke ID in the result packet. After the complete token, there is a delimiter token (EC) and the serial number follows "T000000000059," which actually matches the serial number printed on the side of the switch!

ID	NO	IV	NAME	PS	TYPE	CLASS
ID	CONTEXT					
00	UNIVERSAL					
	NO		OBJECT			CLASS
	01	NODE CONTROL (DEVICE CONTROL)				01
		IV	NAME	PS	TYPE	
		s	serial_#	R	string	
		h	system_addr	R/W	data	
		a	mac_addr	R/W	data	
		g	group_addr	R/W	data	
		n	manuf_name	R	string	
		w	power	R	Boolean	
		l	on_offline	R/W	Boolean	
		o	context_list	R	data	
		i	setup	R	numeric	
		u	user_feedback	R/W	numeric	
		p	product_name	R/W	string	
		f	configured	R/W	Boolean	
		r	controller_present	R/W	Boolean	
		G	default_group_addr	R/W	data	
		N	repeat_enable	R/W	Boolean	
		t	config_master	R/W	Boolean	
		H	report_header	R/W	data	
		A	dest_address	R/W	data	
	02	CONTEXT CONTROL				02
		IV	NAME	PS	TYPE	
		o	object list	R	data	
21	LIGHTING					
	NO		OBJECT			CLASS
	01	CONTEXT CONTROL				02
		IV	NAME	PS	TYPE	
		o	object list	R	data	
	02	ANALOG CONTROL (LIGHT LEVEL CONTROL)				07
		IV	NAME	PS	TYPE	
		c	current_value	R/W	numeric	
		s	saved_value	R/W	numeric	
		r	step_rate	R/W	numeric	
		S	step_size	R/W	numeric	
		f	feature_select	R/W	numeric	

Figure 6: *A typical light-control module would have two contexts, four objects, and twenty-jive Instance Variables.*

USING THE SERIAL NUMBER

Since the serial number is known from the manufacturer's label on the device, it provides a good way to exclusively communicate with this unit for set-up purposes. Normal communication uses the device address after it is set and the device is configured.

However, we will send a broadcast message using the conditional invoke APDU type and ask for the house code in return if the serial number condition is met. The packet this time is a little longer due to the 1%-character serial number and the extra bytes required to form the conditional expression. Note the house code is an array value and must be dealt with using the methods and delimiters for handling arrays.

The packet to get the house code from the module with the serial number equal to "T00000000059" is:

0F 00 00 00 00 01 00 05 00 50 F0 00 01 56 73 EC EC 54 30 30 30 30 30 30 30 30 30 35 39 F7 44 68 F8.

The control byte is the same as before (OF), the destination address is the system broadcast address (00 00 00 00) and the source address is our address (01 00 05 00). The NPDU header (50) is the same, the APDU header is now the conditional invoke type (FO), and we are dealing with the universal context (00) and node control object (01). The CAL message begins with the IF token (56), the serial_# IV (73 "s"), the Equal token (ES), a literal token (EC), and the serial number. The begin token (F7), the Get **Array** method (44), the house code IV (68 "h"), and finally the end token (F8) wraps it up. Simplified-if the universal context object 1, **serial**_# is equal to "T00000000059," then get the array value of the house code. The response packet is:

OF01 00 02 00 01 00 05 00 50 D4 FE F4 32 F6 00 05.

We actually played a sneaky trick on the module. We asked for the house code, which it dutifully sent, but we then ignore the CAL portion of the packet and get the source address, which additionally gives us the unit code without sending another packet!

SUMMARY

I have known about CEBus for the past five years and about six months ago began developing a CEBus product. It was difficult at first because of the broad base of information necessary before you can actually do anything. This article includes a healthy mix of the things that gave me trouble or were hard to find and decipher from reading IS-60.

Good luck on your CEBus project.

Peter House is an applications engineer with Intellon Corporation, a manufacturer of the spread-spectrum carrier components used to implement CEBus on RF and PL media. He may be reached at 71773.2775@compuserve.com.

SOURCES

The EIA CEBus Standard IS-60 is available from:
Global Engineering Documents
1990 M St. N.W., Ste. 400
Washington, DC 20036
(202) 429-2860
Fax: (202) 33 1-0960

The CEBus dimmer module, relay module, serial computer interface, and the module's technical reference manual are available from:
Home Automation Labs
105 Hembree Park Dr., Ste. H
Roswell, GA 30076
(404) 442-0240
Fax: (404) 410- 1122

IRS

416 Very Useful
417 Moderately Useful
416 Not Useful

The Blind Robot

An X-10 Miniblind Automation System

DESIGN CONTEST WINNER

HERBERT **McKINNEY,** JR.

For Herb, home automation includes not only control of artificial lighting, but also control of natural lighting. The X-I 0 Miniblind Automation System offers individuals the possibility to reach beyond the confines of their home and stop the impact of a blistering summer day.

Did you ever wish you could control the light blazing through your skylights on a summer afternoon? That collimated beam sears the plants, nullifies the air conditioning, and slices anything that passes through it like a carbon-dioxide laser. Wouldn't it be nice to have the same command of your home's natural lighting that you have of its artificial lighting?

This project was conceived from just such need.

My wife Kim loves interior designing and, as a result, things get moved around from time to time. She decided the entertainment étagère should be moved from its old location and centered on the large window in our living room. She claimed it would balance the room and back light the figurines in the cubbyholes.

The rearrangement did exactly that as well as creating a magnificent light sculpture! Unfortunately, it introduced a contrast problem for daytime TV viewing and rendered the window's two miniblinds nearly inaccessible.

Operating the blinds became a dreaded task which involved scaling the furniture. To solve this problem, one option was a commercial blind motor. Rocker-switch operated, it offered little more than manual control and sold for around $300.

Our other option of leaving the blinds permanently closed solved the problem of scaling furniture, but left us without our recently acquired backlighting and light sculpture.

Dissatisfied with those choices, I pondered a "techno-cure" that would address all problems involving the blinds, including those throughout the rest of our home. At this point in time, we were making rounds twice a day to open and close them all.

My efforts to eliminate this chore, ultimately coalesced in the X- 10 Miniblind Automation System or. if you'd rather, XMAS, the blind robot. Photo 1 shows the final prototype and Figure 1 illustrates the simplicity of the system.

XMAS has control circuits and a drive motor which fit within the blind's header assembly. An adapter unit connects to an

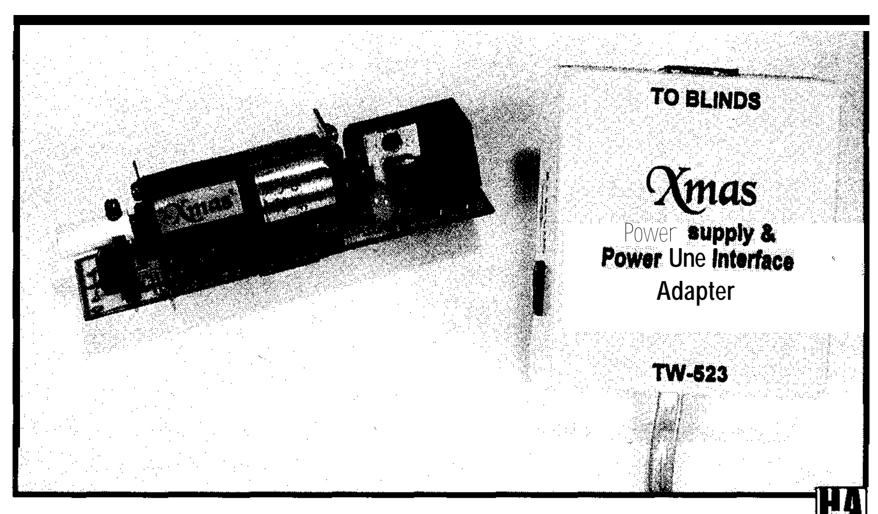


Photo 1: *The final prototype was constructed on a single-sided PCB.*

X- 10 interface module (TW523) and power supply. A modular cable connects the adapter and blind units. The cable may be concealed in a traditional installation manner along baseboards or run through walls to outlets in the window sills for a more professional installation. Up to 256 units may be connected with each unit having a unique address or up to eight units may be grouped into one unique address.

XMAS operates almost as a lamp module. It interprets X- 10 on, off, bright, and dim commands as open, close, up step, and down step, respectively. There are 16 stages between full up and full down. Additionally, the closed position (i.e., off) is jumper selected between up or down. Control and programming of XMAS units can be supplied by virtually any controller capable of transmitting X- 10 commands.

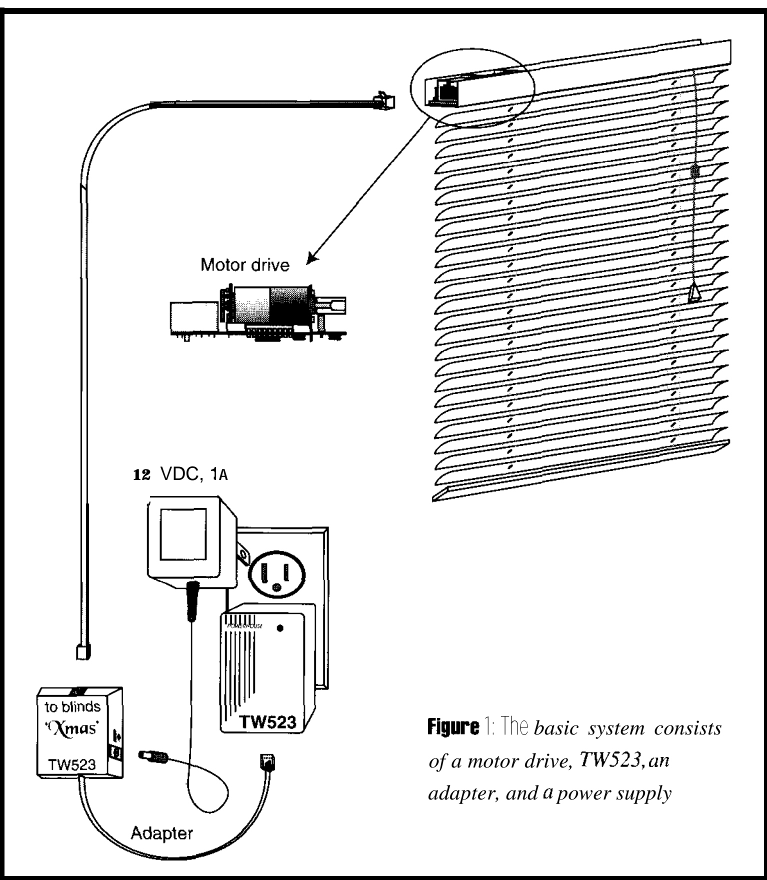


Figure 1: *The basic system consists of a motor drive, TW523, an adapter, and a power supply*

THE DEMON SEED

The humble beginning of XMAS was as elementary as a surplus motor and a VCR load-motor driver. Life was easy.

XMAS evolved from my knack for taking something that is extremely simple and making it much more complicated. XMAS needed to be X- 10 controlled and considerably smaller. It also had to be totally manufactured in my workshop.

After some thinking, I concluded that the primary goals for XMAS were that it be cost effective, easily installed, universal, and retrofittable. Twice, I completely designed it in my mind-ach time allowing ample time for the idea to pass as a silly notion. But, after the code was about half done, I finally pulled out the stops and put it to the drawing board. Kim wanted it next week. Sound familiar?

THE MECHANICS

As Figure 2 demonstrates, the drive motor attaches to the blind's actuator rod with a coupler suitable to the model of blind. A set screw, which secures the coupler to the motor shaft, also actuates the limit switches.

The motor is a 16-mm, 6-V, 15,200-RPM, ⅜-W unit with an extended rear shaft. To this shaft, I attach a photo-reflective disk and sensing PCB to count revolutions. Coupled to the front shaft is a 1670: 1 gearhead. The complete assembly develops an intermittent torque of 14.2 oz.-in. and a continuous torque of 7 oz.-in. This easily exceeds the load of two of the largest blinds I have tested (60" x 72") coupled together end to end.

The main PCB is sized to 3" x 0.94". This form-factor just squeaks in an installation of the smallest of the headers I could find from manufacturers' data, which included 18 popular blinds from many manufacturers. The 3" length fits all but custom-made headers. Many manufactures have standardized on 1" header width or a metric approximation, but I found a lot of variation here.

By using miniature, SOP, and surface-mount devices, the board accommodates all the parts and still offers axial alignment between the motor shaft and actuator rod in the smallest header. The motor and RJ-11 jack placements are relatively fixed and occupy 52% of the board space. So, the remaining components are placed for the tightest fit that routes without DRC errors.

I managed to keep all components on grid, albeit a small one. The mounting tabs of the RJ- 11 connector and forward motor support provide

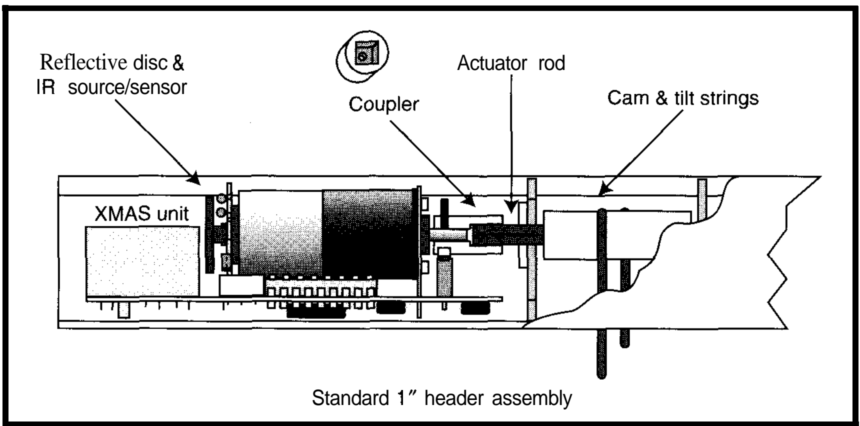


Figure 2: *First, the worm and sector (drive) must be removedfrom the stock header. This is usually a pop-in plastic assembly in newer blinds. The XMAS unit then slides into the header and mates with the actuator rod.*

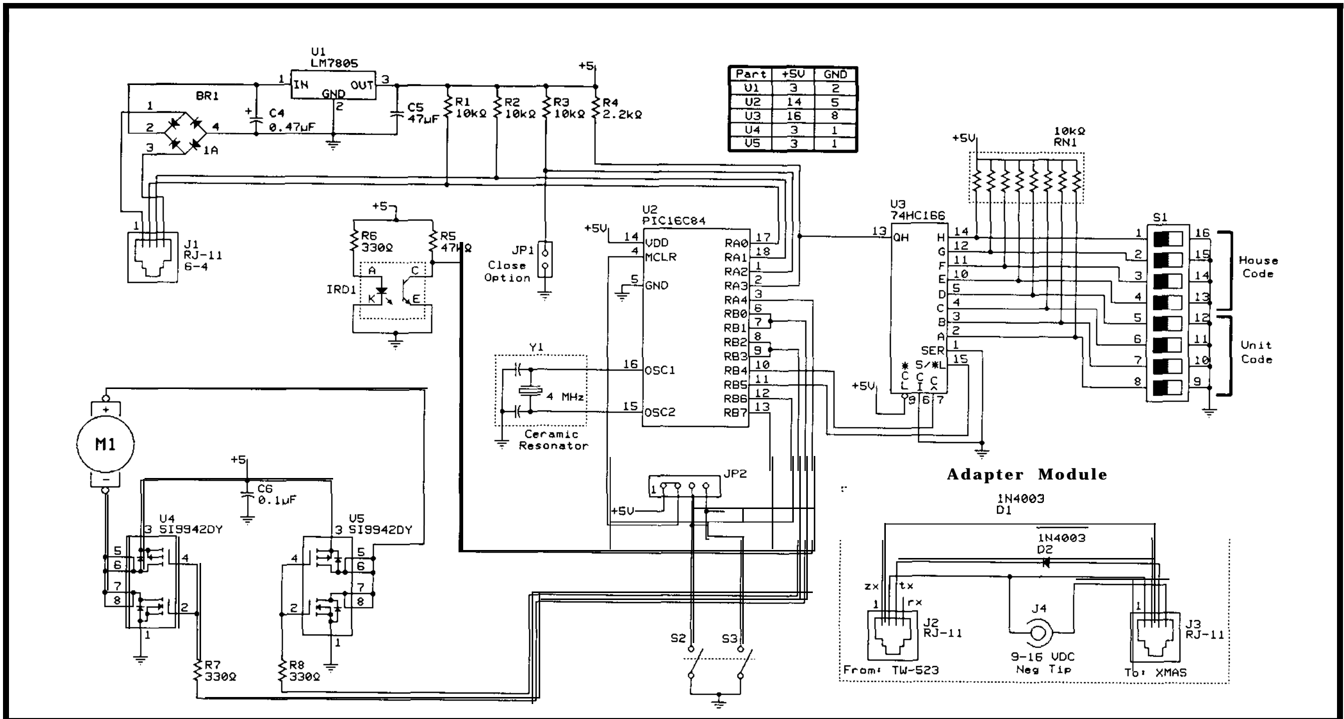


Figure 3: *Thanks to small outline packaging, this circuit (excluding the adapter module) fits onto a 2.8 square inch PCB*

stand-off for the bottom-layer components. There is a 5-mil Lexan sheet beneath for additional protection. I use 0.031″ or 0.062" FR-4 material in larger headers that may need motor alignment.

THE ELECTRONICS

The processor is a Microchip PIC16C84 clocked by a 4-MHz ceramic resonator. Since the PIC16C84 has been covered in prior issues of *INK*, I'll go straight into the details of this application.

The design takes advantage of the PIC's in-circuit programmability. The applicable pins may be accessed after assembly through a jumper header. The EEPROM lets me program the controller and revise firmware on a completed unit without having to remove the chip. This convenience, coupled with the small footprint of the SO-18 package, makes the PIC16C84 the perfect controller for the job.

Figure 3 provides a schematic. The RA port accommodates the ZERO CROSSING, DATA IN, the revolution counter (RTCC), and CLOSE option signals. The specific house and unit codes are set by an 8-bit DIP switch (see Table 1) and read serially through a 74LS166 shift register by RA3.

The 74LS166 (SO-16) occupies Park Place real estate, but it liberates five I/O pins for needed functions. RB 0:1 and 2:3 are motor control bits paralleled to increase drive current for future motor driver improvements. (Note: The original driver was a BAL6686, available only in small quantities. It's a 9-pin, SIP, SOP IC used in RC servo motors.)

I'm pleased to say the board has already been updated to accommodate two Siliconix "Little Foot," dual-complementary, power MOSFETs. This switch involved only minor layout changes on one end of the PCB and greatly improved performance.

Listing 1: *CFG_PINS determines which pins are ZX and DATA.*

```
CFG_PINS call  LILY-10 µs
         sb    PIN17          ;Wait for a quiet cycle
         snb   PIN17          ;with positive going ZX
         sb    PIN18          ;(i.e., both HI)
         jmp   CFG_PINS
chk_lo   nop
         jnb   PIN17,hav_lo   ;wait for 1st low to
         nop
         jnb   PIN18,hav_lo   ;come along (either pin)
         call  DLY_10 µs
         jmp   chk_lo
hav_lo   mov   count,#10      ;wait ~ 5 ms longer to insure
:loop    call  DLY_500 µs     ;data bit time has past in case
         djnz  count,:loop    ;data is coincident with ZX
                              ;(the spec. could allow this).
         jnb   PIN17,:pin17   ;The one that's still
         nop
         jnb   PIN18,:pin18   ;low is LX.
         jmp   CFG_PINS

:pin17   mov   ZX_mask,#01b   ;Zero Crossing = Pin 17
         mov   Din_mask,#10b ;Din = Pin 18
         ret

:pin18   mov   ZX_mask,#10b   ;Zero Crossing = Pin 18
         mov   Din_mask,#01b ;Din = Pin 17
         ret
```

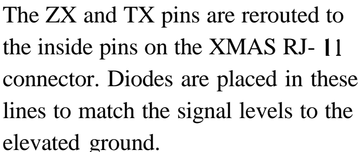


Figure 4: *The main loop is mostly a straightforward interpretation of the X-10 protocol as outlined in the TW523 data sheet.*

Bits 4 and 5 are CLK and LOAD, respectively, for the shift register. Bits 6 and 7 read the LIMIT switches (normally open) and are connected to the programming header as well. NCLR is jumped to VCC through the programming header. I had to forego the recommended ESD protection on /MCLR due to lack of board space. However, with an awareness of this, in concert with reasonable handling, it presents no problem. All rebukes acknowledged!

Overall power for the system is supplied by a 9–16-VDC wall module. Input power to XMAS is wired to the outside terminals of the RJ-11 connector. A bridge was added to allow for polarity reversal after which it is further regulated by a simple 7805 circuit.

The ZX and TX pins are rerouted to the inside pins on the XMAS RJ-11 connector. Diodes are placed in these lines to match the signal levels to the elevated ground.

A software routine determines which inner RJ-11 terminals are ZX and D_in. This scheme allows for straight- or reverse-wired modular cables and a variety of AC/DC converter options. However, it requires a properly wired adapter for the TW523 and DC converter. This seemed to be a worthwhile tradeoff.

To pacify the inspectors, I specify a maximum of eight units sharing a unique address. This restriction is due to a limitation of the wiring. Even though the recommended supply is a power-limited source, it can be easily replaced with a heftier one. Considering a 100% demand factor for these common units, the number should be limited so that the ampacity of the branch wiring is not exceeded.

For a permanent installation in or near a window, 22–24-AWG telephone hook-up wire is highly recommended. This standard allows up to eight units to share a unique address. When using only 26-AWG modular cable, the maximum number of common address units is reduced to four.

One XMAS unit draws 62 mA under a normal load and 220 mA under a maximum, stall-condition load. Fortunately, about the only way to stall this hummer is on the motor shaft itself through a bearing seizure or such. Stalling from the gearhead end produces gear failure. This test is unnecessary. However, it yields about $40 worth of catastrophic failure data for those who feel they absolutely need it!

THE SOFTWARE

To begin, let me confess I've never been accused of generating tight code. I welcome any criticism that advances my capabilities.

Because of having primarily a hardware background, I expected that coding would be more difficult, especially since this was my first PIC project. I chose the Parallax tools to

take advantage of my residual 80*xx* assembler experience, which expedited the task.

As the flowchart in Figure 4 indicates, the software consists of a main loop which lies in wait for a start code from the TW523. It then snares and interprets house, unit, and function codes, and subsequently directs calls to peripheral control routines. A multifunction interrupt routine initially calculates step sizes, then monitors motor movement and effects limit stops.

The **INIT** routine is a little more involved than the flow diagram indicates. After initializing INTCON, OPTION, port direction registers, and the variables, the **INIT** routine calls CONFIG_PINS (Listing 1). CONFIG_PINS determines which of pins 17 and 18 are the ZX and D_in signals for later use in the **BIT-FETCH** routine.

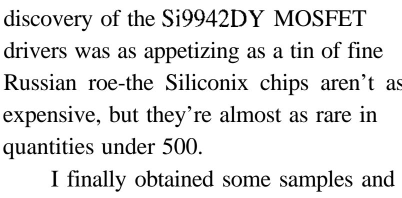
BIT_FETCH is the routine that reads the TW523. A call to GET_ADDR reads the local HC/UC serially from the shift register into **Paddr**. **It** is later compared with the TW523 received code. **INIT** then runs the motor between the high and low limits while accumulating the number of revolutions betwixt the twain with the RTCC using the RT interrupt (the revolutions accumulate in the SVC_I**NT** routine).

RTCC rollovers are stored in the upper byte of **Range while** RTCC leftovers are placed in the lower byte of **Range**. **Range is** then divided by 16 to obtain StpSiz, which is later loaded back into the RTCC to generate the interrupt that stops the motor after each step of bright or dim.

INIT has already read the JP 1 jumper to decide which way to drive the motor to the first (open) limit. After reversing the motor, it leaves the blind in the closed position when the second limit has been found. Notably, the routines MTR_U**P** and MTR_DN always correspond to bright and dim, respectively, but on and off depend on the JP1 condition. On is the center, open position and off is selected between the up or down position with JP1.

CAVIAR & CAVEATS

From the start, I searched for a small-outline motor driver since board space was so limited. My eventual discovery of the Si9942DY MOSFET drivers was as appetizing as a tin of fine Russian roe-the Siliconix chips aren't as expensive, but they're almost as rare in quantities under 500.

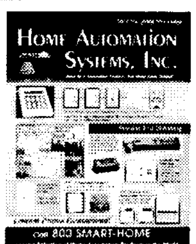


I finally obtained some samples and performed the upgrade. The pair of SO-8s significantly reduce motor run-on after removing the drive signal. The free-wheeling diodes, coupled with the fact that the low-side MOSFETs can conduct during the motor's off state, effectively produce automatic braking.

The improvement was so dramatic that I removed a call to the **BRAKE** routine that previously terminated the **STEP** function. This enabled the main loop to run nearly 200 ms faster in the **STEP** mode. The result was more clearly defined steps and almost a three-fold increase in the continuous step rate.

I chose not to incorporate the X-10 all-units-off and -on commands into my personal units. I may include those commands in the future strictly for compatibility. When CEBus technology stabilizes and miniaturizes (hopefully), I will then

HOME AUTOMATION IS HERE TO STAY!



Get Your Copy of The Best Source of Home Automation Products Absolutely Free.

Largest Selection of Home Automation products in the World

Call 24hrs for Free 64 page Color Catalog

800-SMART-HOME

(800-762-7846)

Hundreds of hard-to-find home automation and wireless control products. Computer control of your home, security systems, surveillance cameras, infra red audio/video control, HVAC, pet care automation, wiring supplies and much more

HOME AUTOMATION SYSTEMS, INC.
151 Kalmus Dr. Ste.M6 Costa Mesa,CA 92626
Questions 714-708-0610 Fax 714-708-0614

#212

What Do You Need To Be a WINNER In the Fast-Growing Home Automation Industry?

Find out when you join the Home Automation Association (HAA)!

Clip and fax today for more information.

Name
Title
Company
Address
City
State/Zip
Phone
Fax

Home Automation Association
Fax: 202/223-9569
E-mail:
75250.1274@compuserve.com
Voice: 202/223-9669

CCI295

#213

endeavor to make this now-simple affair yet more complicated.

Although I have not actually tested 256 of these units connected to *x* miles of cable, I suspect that the system succumbs to the same pitfalls as many distribution schemes. No doubt, cable capacitance eventually wins over rise times. Therefore, the number of units that can be connected to the same TW523 is not guaranteed.

Last, but not least, what can I say? When the power goes off, you're just plain outta luck!

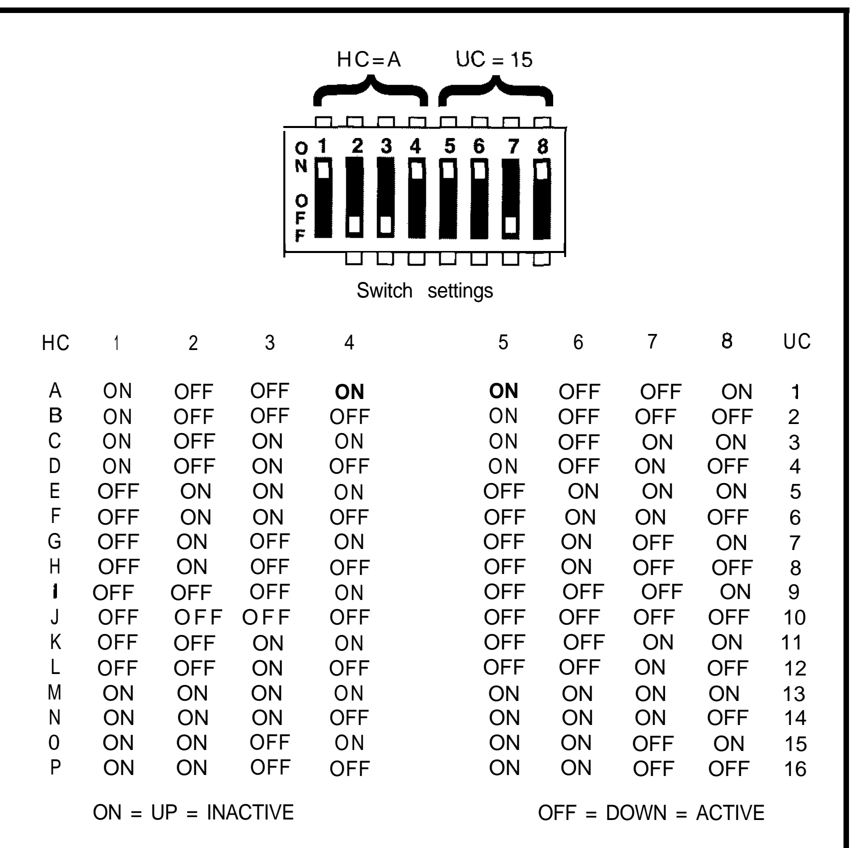
AND TO ALL A GOOD NIGHT...

I initially tried doggedly to dismiss this XMAS idea as cornier than The Clapper, but climbing speakers to close the blinds had become untenable. After contemplating other possible arenas for XMAS such as office buildings, schools, passive solar control, green houses, hospitals, and homes of handicapped individuals, I continued my quest. At approximately $120 per blind (excluding power and control), we consider our dilemma totally resolved.

Perhaps, The Clapper isn't so corny after all!

For now, I'll rest well knowing that XMAS defends our privacy "as visions of sugar plums dance in my head.. ."

Herb McKinney is a former Hewlett-Packard service engineer who currently owns Multi-Technics, a small service consulting and engineering business. He enjoys working with all forms of automation and process control. He may be reached at 75227.2753@compuserve.com.



HC	1	2	3	4	5	6	7	8	UC
A	ON	OFF	OFF	ON	ON	OFF	OFF	ON	1
B	ON	OFF	OFF	OFF	ON	OFF	OFF	OFF	2
C	ON	OFF	ON	ON	ON	OFF	ON	ON	3
D	ON	OFF	ON	OFF	ON	OFF	ON	OFF	4
E	OFF	ON	ON	ON	OFF	ON	ON	ON	5
F	OFF	ON	ON	OFF	OFF	ON	ON	OFF	6
G	OFF	ON	OFF	ON	OFF	ON	OFF	ON	7
H	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF	8
I	OFF	OFF	OFF	ON	OFF	OFF	OFF	ON	9
J	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	10
K	OFF	OFF	ON	ON	OFF	OFF	ON	ON	11
L	OFF	OFF	ON	OFF	OFF	OFF	ON	OFF	12
M	ON	ON	ON	ON	ON	ON	ON	ON	13
N	ON	ON	ON	OFF	ON	ON	ON	OFF	14
0	ON	ON	OFF	ON	ON	ON	OFF	ON	15
P	ON	ON	OFF	OFF	ON	ON	OFF	OFF	16

ON = UP = INACTIVE   OFF = DOWN = ACTIVE

Table 1: *Cost considerations and layout constraints made a DIP switch and negative logic compulsory* **for** *address setting. The address is encoded by the switch settings. After serializing, it assumes the correct order for direct comparison with the received code. This eliminates the need for a software conversion table.*

X-10 Technical Note: Two-Way, Power-Line Interface Model **#523**
X-10 (USA), Inc.
185A LeGrand Ave.
Northvale, NJ 07647
(201) 784-9700

REFERENCES

PIC16C84, PIC16C84 Reference Manual, and **DS30081B**
Microchip Technology, Inc.
2355 West Chandler Blvd.
Chandler, AZ 85224
(602) 786-7200
Fax: (602) 899-9210

PIC16C*xx* Development Tools
Parallax, Inc.
3805 Atherton Rd., #102
Rocklin, CA 95765
(916) 624-8333
Fax: (9 16) 624-8003

SOURCES

Digi-Key Corp.
701 Brooks Ave.
P.O. Box 677
Thief River Falls, MN 56701-0677
(800) 344-4539
Fax: (218) 681-3380

1616E006ST123/16A1670:1 gearhead motor
Micro Moe
742 Second Avenue South
St. Petersburg, FL 33701
(8 13) 822-2529
Fax: (8 13) 82 l-6220

Mouser Electronics, Inc.
12 Emery Ave.
Randolph, NJ 07869
(800) 346-6873
Fax: (201) 328-7120

Siliconix **Si9942DY**
Rep, Inc.
Temic Group
P.O. Box 728
Jefferson City, TN 37760
(615) 475-9012
Fax: (6 15) 475-6340

BAL 6686 **(#T39900)**
Futuba Corp.
4 Stedebaker
Irvine, CA 927 18
(714) 455-9888

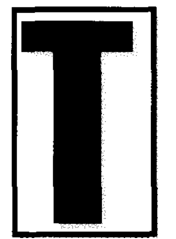
I R S

419 Very Useful
420 Moderately Useful
421 Not Useful

A Learning Remote-Controlled Speaker Selector

SCOTT **HEISERMAN &** CLARK **ODEN**

Scott and Clark set out to find remote-speaker selection without replacing their current stereo system or spending too much money. The ultimate solution: an add-on component with an innovative intrared learning algorithm and a highly integrated microcontroller.



The Remote Controlled Speaker Selector (RCSS) addresses the challenge of creating a convenient, multiroom listening environment for the home. Most stereo systems have manual A/B speaker selection which provides music to one of two rooms or both rooms simultaneously (A+B). If that's not enough, an external speaker selector can be added easily.

But what if you're outside in the pool and the urge to swim laps to "Born in the USA" suddenly grabs you? In this scenario, you must go inside, negotiate a polished kitchen floor with wet feet, and switch the stereo to play over the pool speakers-not exactly the dream of home automation.

Some high-end systems address the problem of multiroom listening by using proprietary modulation schemes. These schemes multiplex audio and two-way data over user-installed coax to each room. The problem with this solution is that a perfectly good stereo system has to be replaced.

Alternatively, a few add-on devices can be used with existing stereo systems in one way or another-some use X- 10, infrared, or combinations thereof. However, these systems are fairly expensive, often compromise amplifier safety by switching only one side of the output, or lack user feedback, which is essential in remote switching.

DEVICE DESCRIPTION

The RCSS is an add-on home-stereo component designed for loudspeaker selection from virtually any infrared (IR) remote controller. An innovative learning algorithm and high-integration microcontroller make the RCSS "smart" as well as inexpensive with its low parts count.

The RCSS can be used with off-the-shelf IR repeater systems for separate room-speaker selection. This lets a listener select speakers from whatever room they are in without having to physically make a selection at the amplifier or receiver location. Since most existing stereo systems can already be remotely controlled with an off-the-shelf IR repeater, the RCSS adds the speaker-select function that most stereo systems lack. With the RCSS, the user obtains multiroom listening convenience while retaining their existing audio equipment. A diagram of the RCSS is provided in Figure 1.

SYSTEM OVERVIEW

Specific highlights of the RCSS include:

- Learning algorithm-The RCSS offers maximum flexibility. It can be controlled by virtually any IR remote controller, regardless of manufacturer. Low-cost generic IR remotes can be used for selection control.
- Four speaker pairs-Four independent speaker pairs can be selected with the RCSS.
- Manual operation-A front-panel push button provides manual selection of individual speaker pairs as well as dual-pair combinations for two-room listening.
- Indicator lights-Four green LEDs give visual status of speaker selection(s). A red LED marks learn status (on = learn mode). The red LED flutters on initial powerup to show that the RCSS needs programming.
- Confirmation tone-A dual-frequency confirmation tone is sent to the selected speaker pair before the music source is switched in. The

confirmation tone provides an audible indication that the correct speaker pair has been selected.

- Program retention-RCSS remembers the commands it has learned when power is interrupted or the unit is unplugged. A replaceable 10-year lithium coin cell provides power backup.
- Low cost-The cost for the electronic portion of the prototype was under $50.

DEVELOPMENT OBJECTIVES

A major design goal in the development of the RCSS was to make it compatible with most hand-held IR remote controllers—no one needs another remote to add to their collection. And, most controllers have extra, never-used buttons. Thus, developing a device capable of learning and recognizing existing IR controller codes was central. Although a simple sampling method could work, the memory requirements for even a single IR code are relatively large, even with the application of rudimentary compression techniques (INK 29).

IR CONTROLLER CODES

When you push a button on a remote controller, the remote emits a series of infrared bursts. The bursts, which amount to switching a pulse carrier on or off, carry the code corresponding to the button.

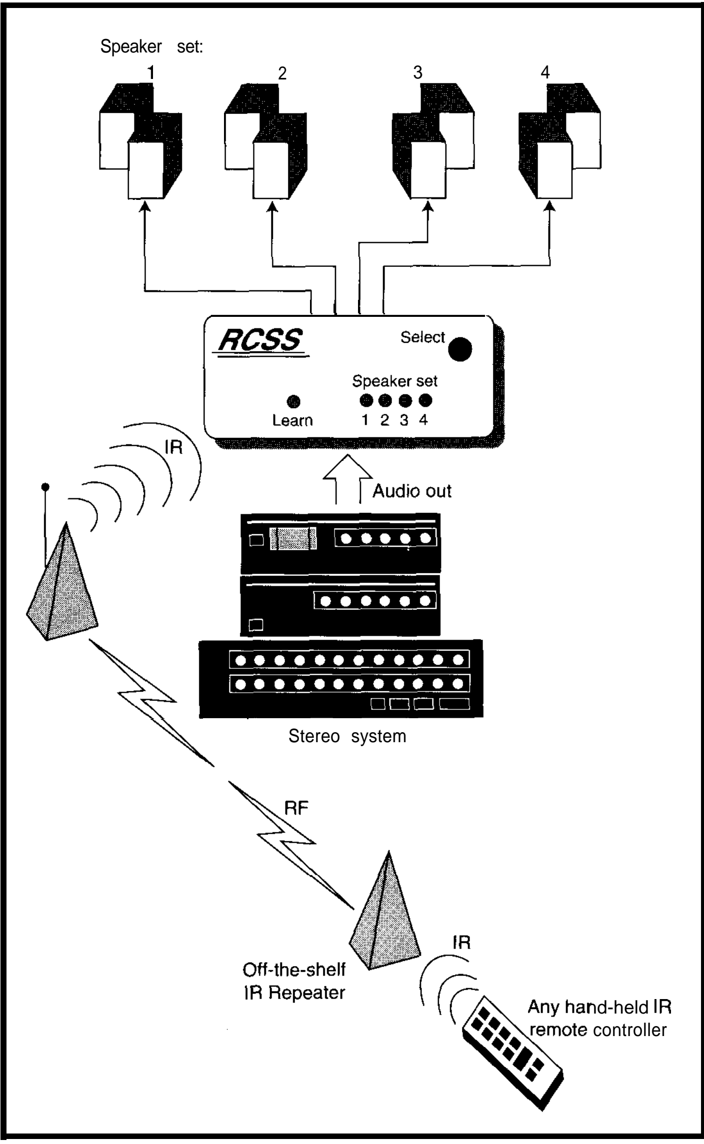


Figure 1: *This conceptual diagram shows how the RCSS can be used in a home environment. Low-cost IR repeater transmitters are located with each speaker pair.*

Pulse-carrier frequencies range from 25 kHz to 60 kHz, with the most common around 38 kHz. The carrier bursts usually last from 0.5 to 2 ms in duration and correspond to a bit in the function code. An entire code sequence may have 12 to 32 bits (or bursts), so the code frame time would be on the order of tens of milliseconds.

Most remotes also have a common sequence marking the start of all the codes they transmit. It essentially acts as a wake up preamble. The modulated information sent by the remote is demodulated by the receiving device into an asynchronous stream of binary highs and lows which generally contains a preamble sequence, manufacturer and device information, and the specific function command.

Manufacturers can and do use different schemes for embedding information in the infrared flashes-there are no industry standards for encoding. Most use some form of pulse-width modulation which conveys bit information according to carrier-burst duration. The bits can be represented by the actual bursts or by the time between them.

Manufacturers also have unique schemes for repeat functions. Say you want to crank up the TV volume. You hold down the Volume+ button. One manufacturer's remote transmits the entire Volume+ command repeatedly, while another sends the Volume+ command once followed by a shorter repeat sequence for as long as you hold down the button. Figure 2 shows the start of a few typical received IR codes.

Though there are undoubtedly countless control codes, with a learning device, it does not matter. For the RCSS, the only thing that matters is that it recognizes a learned button when it is pushed again. To do this, the RCSS has to pick apart and store the necessary elements of a button's code sequence. In general, the code sequences follow these criteria:

- Code sequences always follow the format: preamble, space, code information
- The preamble is at least three times longer than a space
- Space defines the duration for a binary 0 (arbitrarily assigned)
- Space is always the inverse polarity of binary 1s and 0s.

That is, when bits (1 s and 0s) are represented during the times the LED is modulated on, the space between bits occurs when the LED is off. Conversely, if the bits are represented during the times that the LED is off, then the space between bits **occurs** when the LED is on.

These generalizations hold true for the vast majority of infrared codes. In the simplest terms, the RCSS algorithm measures the duration from one transition to the next, producing a count. The count represents both the time that the LED is modulated with a burst and the time between bursts.

The count is stored, another event is timed, and the new count is subtracted from the previous count. From this result, the program determines whether the bit of code information is a 1 or a 0, and the process repeats for succeeding bits.

The algorithm has been developed based upon the following protocol generalizations:

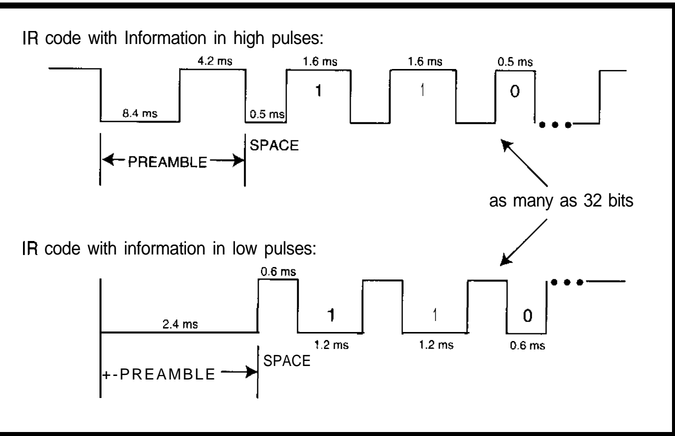


Figure 2: *The typical IR code sequence contains pulse-width modulated data in either the low or high portion of the pulse train. Binary bit information is encoded in the pulse-width variations. Fixed-width pulses correspond to spaces between bit information.*

- The first and second counts, essentially the preamble, don't matter and can be discarded
- The absolute difference between a low and high count following the preamble is significantly greater than 0 for a binary 1 and near zero for a binary 0.

Using these assumptions, the RCSS algorithm produces a compact binary representation of the incoming remote code, regardless of the carrier frequency, the bit rate, or the format for 1s and 0s.

In the program, each absolute difference is checked against a tolerance value for translation into either a 1 or a 0. The bit is then packed into a four-byte holding location. Each IR remote button learned is represented in 32 bits (whether it needs that much room or not) to keep the algorithm simple. Occasionally, more than 32 bits are required, but the majority of controllers operate at 32 bits or fewer.

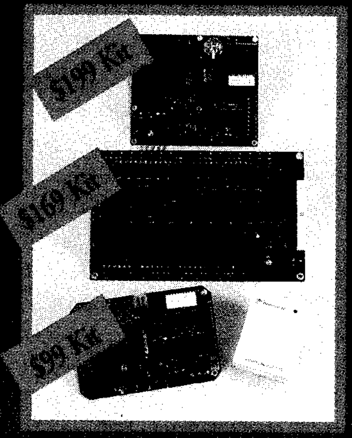
FIRMWARE

The general development approach of the RCSS was to do as much as possible in firmware including switch debouncing, IR signal recognition, confirmation-tone output, and front-panel indication. This approach not only minimizes cost by reducing parts count and circuit-board real estate, but also facilitates the development of an intuitive user interface. The interface is important because most of the time the user would not be within sight of the RCSS. Additionally, the intuitive interface bolsters user confidence in the training process and front-panel operation.

FIRMWARE OPERATION

Figure 3 is an overall flow diagram of the RCSS software. The program starts at the

Home Control is as simple as 1, 2, 3...



1) The Circuit Cellar HCS2-DX board is the brains of an expandable, network-based, control system which incorporates direct and remote analog and digital I/O, real-time event triggering, and X-10 and infrared remote control. Control programs and event sequences are written on a PC in a unique user-friendly control language called XPRESS and stored on the HCS2-DX in nonvolatile memory.

2) The Relay BUF-Term provides hardwire interface protection to the HCS2-DX. Its 16 inputs accommodate both contact-closure and voltage inputs within the range of ±30 V. Its 8 relay outputs easily handle 3 amps AC or DC to directly control solenoids, motors, lamps, alarm horns, or actuators.

3) The PL-Link provides wireless X-10 power-line control to the HCS II system. The PL-Link is an intelligent interface that sends, receives, and automatically refreshes X-10 commands. It works with all available X-10 power-line modules.

Of course, there's a lot more! The HCS II system has phone and modem interfaces, infrared remote control, voice synthesizers, and much more. Whatever your application, there's a configuration to accommodate it.

CIRCUIT CELLAR, INC.
4 Park Street, Vernon, CT 06066
(203) 875-2751 • Fax (203) 872-2204

#214

beginning of ROM (location $0200) with a series of qualified initializations. The ports are defined, then port A is read. If the lower four bits of port A are cleared-a normally illegal state-thenthe TEMP2 register is loaded with $FF as a first-time powerup flag for use later in the program. Other qualified initializations include clearing the IR code storage locations (CODE), common registers, and count variables. This portion of the program is recycled by different routines to conserve program memory.

After qualified initializations, the computer operating properly (COP) register is reset. This paves the way for a series of bit-level interrogations. First, port A, bit 4 is checked for manual switch closure. If the switch is closed, control is transferred to the **MANSW** routine.

Next, the Learn switch, bit 1 of port B, is checked for closure. If it has been pushed, both the first LED (speaker set 1) and the Learn LED are lit. The program then monitors for IR input. If the Learn switch has not been pushed, TEMP2 is checked for $FF and the Learn LED is toggled if it is $FF.

Finally, bit 0 of port B is checked for IR input. If none is present, the program returns to reset the COP register. It continues this loop until IR is detected at bit 0 of port B.

When port B, bit 0 finally goes low-signaling IR input-program control is transferred to the **READ** routine. The IR data is serially sampled at a 0.1 -ms rate and stored in indexed code RAM locations. After input, control is transferred to the **STORE** routine.

STORE first checks for learn mode. If the learn register, **TEMP2**, is set to 1, 2, 3, or 4, code bytes are transferred to the appropriate storage locations. If not in the learn mode, program control is transferred to the **RECOG** routine.

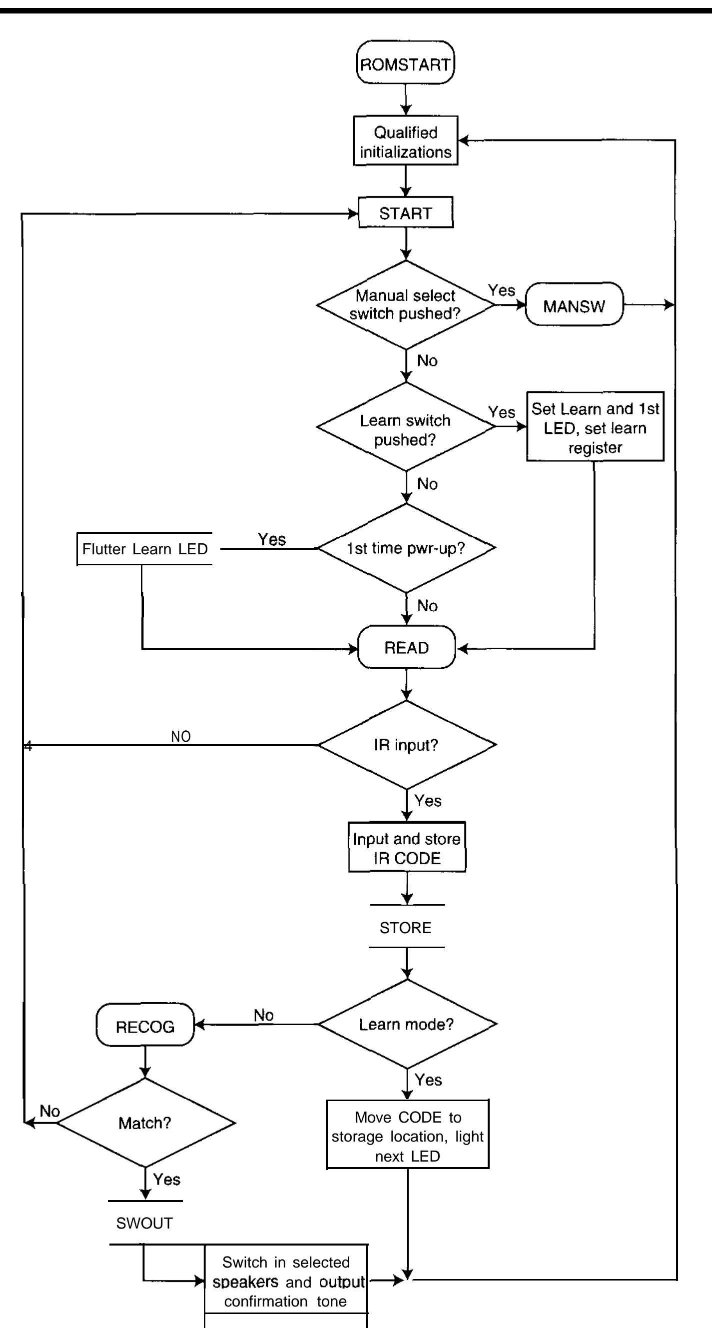


Figure 8: *A modular approach was used in the development of RCSS firmware when possible. This overall flow diagram shows that some routines are recycled to make best use of the 68HC705's tiny 0.5 KB of ROM and 32 bytes of RAM.*

RECOG sequentially compares stored bytes with the code read in. If there is no match, the program returns to the beginning where sequential checks are performed again. If there is a match, control is transferred to the **SWOUT** routine.

SWOUT performs speaker and source relay switching and debouncing, and output confirmation-tone generation. After the switching is complete, the program returns to **START**. ∎

The program fully utilizes the 68HC705 microcontroller to provide IR code learning and recognition as well as an intuitive user interface. All RAM and most of the ROM is used. Real-time interrupts are not used because of RAM limitations and they simply are not needed. The microcontroller operates in the microsecond world, whereas IR codes are in the millisecond domain.

HARDWARE

The hardware components and layout are designed for a high degree of integration, low parts count, and short wiring runs. All components are available from several sources.

The RCSS is designed so that all wiring connections are made at the rear panel, with the front panel of the aluminum enclosure reserved for operating controls and indicators. Construction is by hard wiring, but the circuit board may be removed from the case by unfastening the front and rear panels. This design provides for high reliability while using commonly available components. Figure 4 shows the schematic layout.

POWER SUPPLY

Power enters the RCSS at rear-panel power connector J1, a 3.5-mm phone jack. A 1N4004 diode protects the circuitry from reverse voltage should a power source of opposite polarity be plugged into the rear panel. A 1-µF ceramic capacitor filters the power input, and a MOV provides

surge protection for voltage spikes over 33 V to the voltage regulator (U5). U5 is a 5-V linear regulator in a TO220 package. The 5-V output of the regulator is bypassed by both a 0.1-µF and a 0.01 -µF capacitor.

Battery backup and switchover is accomplished by U4, a Maxim MAX704 supervisory circuit designed for use with microprocessors. During normal operation, U4 simply passes the 5-V supply to the U1 microcontroller V_{DD} pin. However, if the 5-V supply drops below 4.4 V, U4 holds the microcontroller reset pin low and switches the V_{DD} supply to a 3-V lithium battery.

This scheme provides backup power for the microcontroller RAM. Thus, the microcontroller RAM is nonvolatile and its contents are retained in the event of a power outage. Even frequent, short-duration power interruptions do not significantly reduce the battery's life below its expected shelf life.

All other power connections are made to the 5-V regulator output.

MICROCONTROLLER

U1 is a Motorola MC68HC705K1 microcontroller. U1 receives data from the infrared module at port PBO. The Select and Learn switches are read at ports PA4 and PB1, respectively. A 4.000-MHz crystal clocks the microcontroller. Port PA6 is configured as an output to control the input audio-source relay driver. Output ports PA0 through PA3 control the four speaker audio-output relay drivers. Output port PA5 generates a confirmation tone, which is sent to one of four speaker outputs. Output port PA7 drives an LED driver for learn-mode indication. The IRQ line is pulled up (disabled).

Under software control, the microcontroller reads and learns infrared codes, or reads IR codes and selects speaker-pair outputs. U1 also disconnects the audio-source input, generates a confirmation tone, and reconnects the audio-source input when a new output is selected.

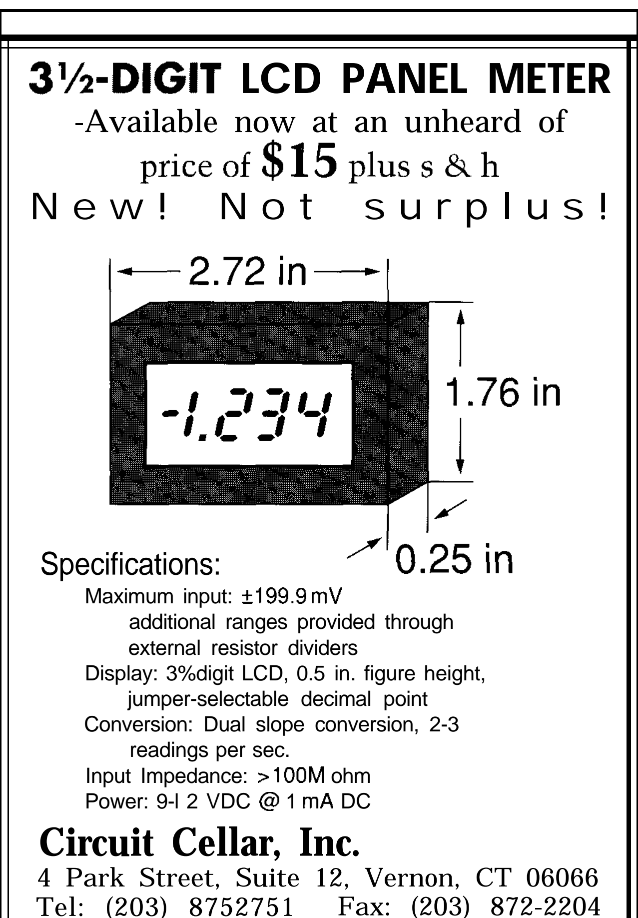
RELAYS AND LEDS

Relays Kl-K6 are all Aromat JW-series relays, which can switch up to 5 A. K1 and K2 are input relays that have two form-C contacts each, thereby enabling the hot and ground signals from both channels of a stereo amplifier to be disconnected during confirmation-tone generation. During tone generation, K1 and K2 select local ground and tone from U1 port PA5 to be sent to the output relays.

K3 through K6 all have two form-A contacts, which switch only the hot (+) signal from each channel input, either on or off. All relay drivers are PNP transistors contained within U2 and U3 transistor packages. The PNP relay drivers are protected from inductive kickback by 1N914 diodes across the relay coils. The output relay driver U2 also drives four green, front-panel LEDs for indicating front-panel output selection. One of the U3 transistors drives the red LED for front-panel indication of the learn mode.

SWITCHES, HARDWARE, AND IR MODULE

The front-panel, momentary push-button switch Select is read by U1 to sequentially select the speaker output from the front panel. The rear-panel Learn switch is read by U1 to put the unit in learn mode.



#215

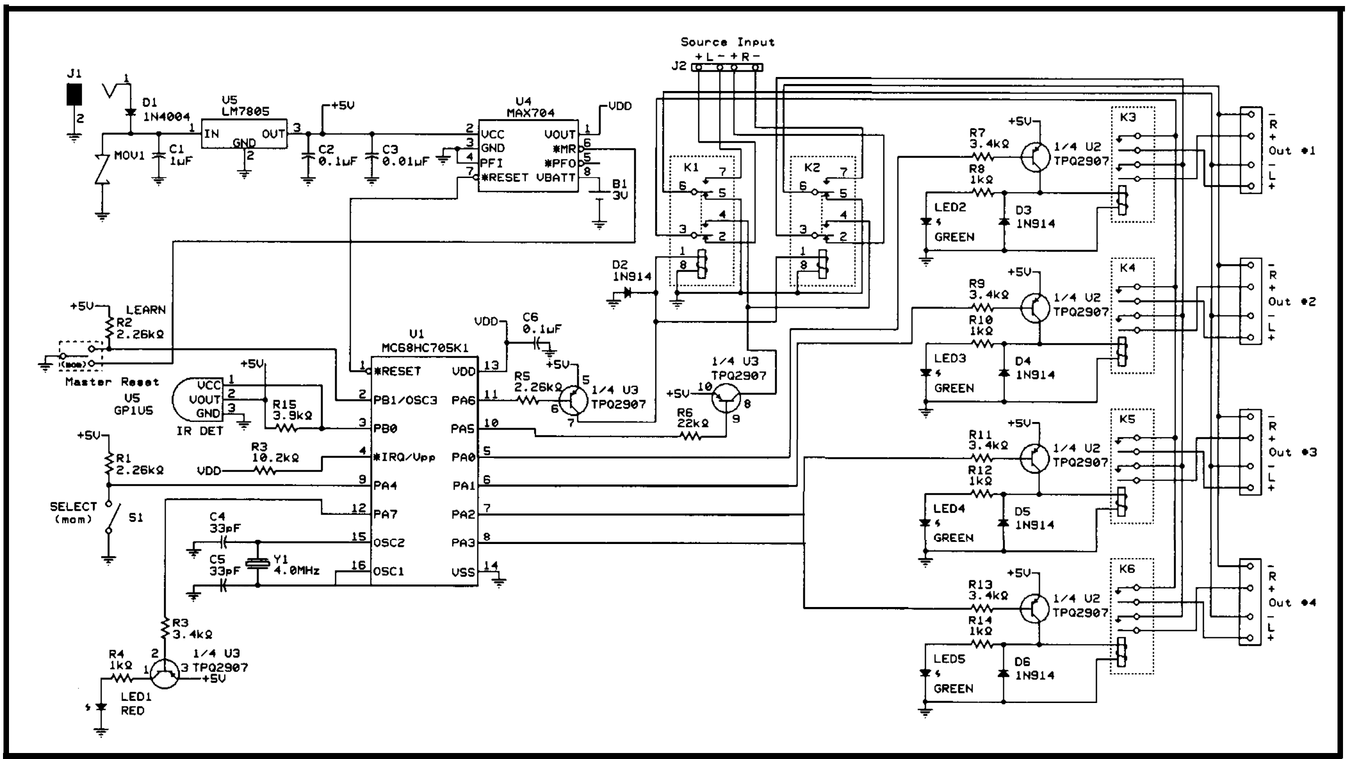


Figure 4: *Hardware for the RCSS was kept to an absolute minimum by performing most tasks in firmware.*

An infrared receiver/demodulator (Sharp GP1U5) on the front panel filters and demodulates the incoming infrared code to be read by U1.

The rear panel has spring-terminal connectors for four speaker pairs (16 terminals). All connections from the rear panel to the perfboard are made with stranded 20-AWG wire. A rear-panel, 4-position, polarized interlocking connector provides connection to the source amplifier. Power is supplied through a 3.5-mm phone jack on the rear panel. The case is black, anodized, extruded aluminum with an integrated card guide for the board.

OPERATION

The RCSS is operated with an IR remote control. To use the RCSS, it must be installed and programmed for the particular system it is to be used with. Up to four speaker sets may be connected. Speaker wiring should be installed before the RCSS is set up. The following sections address installation, setup, and operation.

INSTALLATION

Before making any connections to the RCSS, the lithium backup battery should be installed. Although the backup battery is not required for operation, it enables RAM data retention when normal power is disrupted. This means that the RCSS does not have to be reprogrammed after a power interruption.

Refer to Figure 5 for the connection layout on the rear panel. The easiest way to get the RCSS up and running is to first make connections to the speaker sets and source amplifier on the rear panel, then position the RCSS where it can receive infrared signals. Speaker sets are wired to the rear panel using 16 spring-release terminals. Red is positive and black is negative for the four sets of terminals with the right channel located along the top row. The source amplifier is connected through the interlocking connector (Molex) with pigtails.

Power is supplied through a wall transformer. Plug the 3.5-mm phone plug from the wall transformer into the power jack on the RCSS rear panel, and plug the transformer into a 110-VAC outlet. There is no power switch so the RCSS normally remains on. When power is connected for the first time, the front-panel Learn indicator (red) blinks. The RCSS is now installed and ready for setup programming.

SETUP PROGRAMMING

Figure 5 shows the front panel during the setup programming discussion. The RCSS blinks the front-panel red LED when it has not been programmed. The first step in setup is to decide which remote control operates the RCSS.

Virtually any common IR remote control works. The idea is to pick four buttons on a remote (or remotes) to select among the four speaker set outputs. In many cases, there are some buttons on an existing remote control that are unused or operate a component not used in the your system.

An example of this might be a receiver remote that includes buttons for controlling a same-brand CD player when the CD player owned is a different brand. In this case, the CD buttons can be used to operate the RCSS from the receiver remote. You could also purchase an inexpensive replacement remote for TV or VCR or use four buttons on a remote from a remote-controlled component or TV not being used. Again, most remotes work. Just pick four buttons on any remote.

To program the RCSS, push the rear-panel Learn switch down. This puts the RCSS into learn mode (indicated by the steady illumination of the red LED on the front panel).

Stand several feet away from the unit, point the remote control at the unit, and push and release the four buttons on the remote corresponding to each speaker set in sequence 1, 2, 3, and 4.

Each time a button is pressed, the green LED corresponding to the next speaker set to be programmed illuminates. When all four buttons have been pressed, all front-panel indicators go out. When the red Learn indicator turns out, the unit is in recognize mode. When all green indicators turn off, no speaker sets are selected. By pressing any of the four buttons just programmed, the RCSS selects the corresponding speaker set output. The RCSS will not respond to other IR remotes or buttons.

Note that programming the RCSS must be done under optically quiet conditions. This means that any IR repeaters should be covered or otherwise disabled and no other IR remotes are in use. Also, during programming, if the remote buttons are held down too long, then several speaker sets will be programmed to the same remote button. If this happens, simply reprogram the RCSS by depressing the rear-panel Learn switch and pressing the appropriate remote-control buttons again.

USING THE RCSS

When the RCSS has been programmed, it is ready for use. The RCSS works with commonly available IR remote repeater sets. These sets usually have a transmitter and receiver. They convert the remote control's IR signals to an RF signal, transmit them to a receiver, which then converts the signal back to IR to control the component.

Some repeaters are hard wired. But, regardless of the technology used, the result is the same. A repeater can be placed in any room where secondary speakers are located, enabling remote-control selection of that set of speakers from that room.

When a particular set of speakers is selected by remote control, a confirmation tone is sent to the speaker set. This confirms that the correct selection was made since the user typically cannot view the front-panel indicators on the RCSS. Each time the selection is made by remote, a confirmation tone is sent.

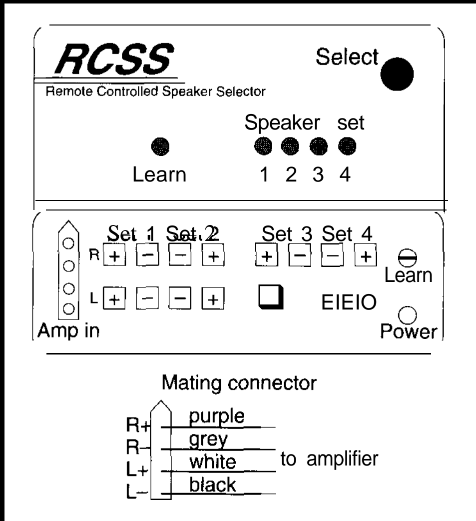


Figure 5: The front panel has LEDs for speaker selection and learn status and a switch for manual selection of speaker pair(s). Rear-panel push terminals are for speaker connection and a molex connector is for the source amplifier. The switch is a momentary, center-off switch. Down invokes the learn mode and up resets the microcontroller.

The Select button on the front panel enables a local speaker-set selection. Each time Select is pressed, another speaker set is selected as indicated by the front-panel indicators. The Select button also enables the user to select any two speaker sets at once. The Select button follows this sequence: 1, 2, 3, 4; 1 and 2; 2 and 3; 3 and 4; 1 and 3; 2 and 4; 1 and 4. The pattern then repeats. There is no confirmation tone when Select is used to select speaker sets.

The infrared detector in the RCSS is quite sensitive and is typically able to read infrared codes from 30' or more. This means remotes or IR repeaters can be conveniently and aesthetically located. The only requirement is that there must be a clear line of sight from the repeater receiver to the RCSS IR detector on the front panel.

CONCLUSION

The RCSS switches four speaker pairs from one stereo source by recognizing unique IR codes from common IR remote controls. Combining the RCSS with an IR repeater enables remote-controlled speaker selection from any location within the repeaters range.

Relay-switching capacity during audio peaks is 5 A, which corresponds to 200 W into 8 Ω. The peak current capacity of closed contacts is much higher, so virtually any power level can be accommodated with no interference to sound quality (low-resistance contacts).

The unit has optional front-panel manual controls and indicators and an internal tone-signal generation to provide user feedback of successful remote switching. A very efficient code-recognition algorithm means a small and inexpensive microcontroller can be used. No exotic parts are necessary for construction, so cost is reasonable.

Scott Heiserman holds an MS in electrical engineering. He currently develops analog and digital modifications and embedded solutions for the FAA. He may be reached at 70671.2773@compuserve.com.

Clark Oden holds a BS in electrical engineering and designs precision time and frequency equipment. He also works with RF, analog, hardware, and software for FAA applications.

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.

SOURCE

A preprogrammed 68HC705 may be ordered for $25 postpaid from:

RCSS Project
10104 St. Helens Dr.
Yukon, OK 73099

I R S

422 Very Useful
423 Moderately Useful
424 Not Useful

DEPARTMENTS

82 Firmware Furnace

92 From the Bench

98 Silicon Update

106 Embedded Techniques

115 ConnecTime

FIRMWARE FURNACE

Ed Nisley

Journey to the Protected Land: With Interrupts, Timing is Everything

All these protected-mode and multi-tasking machinations really do lead to a land of simplified code, at least when it comes to error handlers. Ed beefs up the bug catchers while looking at some interrupt-handling issues.

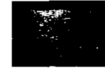
On my time line, it's mid-December and the Pentium FPU firestorm threatens to consume Intel's credibility, if not their future. On your time line, it's late March and you know how the story ends. All I can say now is that I'm glad for my plain old 486DX2. Sometimes the thick edge of the wedge is the place to be!

Just as all programs have bugs, all hardware has quirks. If you never stumble upon the circumstances that trigger a quirk, its presence doesn't matter to you. Knowing that a quirk exists can either help you avoid it or justify buying something else. That may explain why it's so difficult to get errata lists-if a bug isn't mentioned, does it really exist?

This month, we'll reinforce the error handlers that catch our own bugs, then measure a hardware interrupt's response time when it triggers a task switch. The venerable 8259 Programmable Interrupt Controller and all its LSI progeny have an interesting, well-documented quirk that most folks have never encountered; you'll get the story here!

IN CASE OF EMERGENCY...

Ever since we first flipped into 32-bit protected mode, a simple error handler has watched for protection



Listing 1—The unexpected error *handler deals with **interrupts** that are not caught by any other handler. The error handler **setup** code fills the **IDT with** 256 **interrupt** gates aimed at these 256 stub routines. Each **stub** pushes the Interrupt ID on **the stack** and **task** switches info the handler by **executing** a FAR CALL containing its **TSS selector.** Although fhe stubs include an **IRET** fo return control **to the** failing fask, the **FFTS** error handler simply displays an error dump and **halts the** system.*

```
NUM_TASK_VECTORS = 256        ; all possible interrupts

        CODESEG
        ALIGN     2             ; get a nice offset
        PROC      ErrTaskVectors

@@ID    =         0
        REPT      NUM_TASK_VECTORS
        DB        06Ah          ; PUSH immediate byte, MSB = 00h
        DB        @@ID
        DB        09Ah          ; FAR CALL with imm seg:offset
        DD        0             ; offset is not used here
        DW        TSS_ERRORS    ; seg causes task switch
        IRET                    ; return from interrupt (ha!)
@@ID    =         @@ID + 1
        ENDM

        ENDP      ErrTaskVectors

TASK_VECT_SIZE = $ - ErrTaskVectors ; total length of all stubs
TASK_VECT_STEP = TASK_VECT_SIZE/NUM_TASK_VECTORS     ; stub size
```

violations. Without the support of the CPU's multitasking hardware, however, it's difficult to write an error handler that doesn't mess things up while attempting to display an error message.

As a result, the only indication of an error was a cryptic pattern on the Firmware Development Board and parallel port LEDs identifying the failing instruction. While that may be better than real-mode pinball panic or a system freeze, we can do much better using separate tasks for the error handlers. You knew multitasking was going to come in handy for something, didn't you?

Figure 1 shows the sequence of events after the CPU detects an error in protected mode while running a task. If the IDT entry corresponding to that error contains an interrupt or trap gate (the other choice is a task gate, which we'll discuss shortly), the CPU pushes the current EFLAGS, CS, and EIP registers onto the stack. Some errors also produce an error code to help identify the problem, which the CPU pushes atop EIP, rendering a simple **IRET** impossible. Figure 3 in *INK 50* tabulates the predefined interrupts, their types, and whether they produce an error code. (Note that there is a table of acronyms at the end of the article for those who didn't quite follow the past few sentences.)

The interrupt or trap gate directs the CPU to a stub routine that pushes the interrupt ID number. Without that value on the stack, the handler cannot tell which interrupt activated it. The alternative is 256 separate interrupt handlers, which seems excessive even to me. Listing 1 shows the macro that generates 256 stubs leading to our new error handler.

Each stub includes a synthetic **FAR CALL** with the TSS_ERRORS selector in the segment position. That selector corresponds to the TSS of the error-handler task. The CPU reacts to this **FAR CALL** by storing the failed task's state in its TSS and task switching to the error handler. As always, the CPU loads a new state from the incoming TSS, ensuring that all the registers are safe from harm and the new stack is entirely separate from the old one.

If you thought task switching was complex last month, hold onto your keyboards. Figure 2 shows the situation just after the task switch. The stub's **FAR CALL** triggers two new actions during the task switch: the CPU stores the failed task's TSS selector in the error handler's TSS

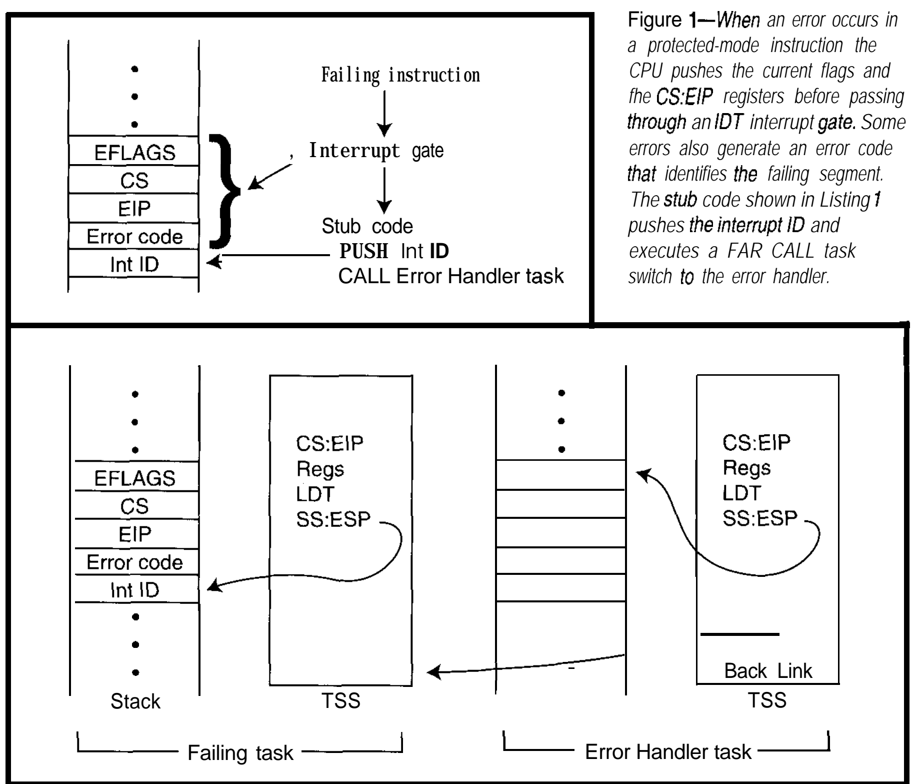


Figure 1—*When an error occurs in a protected-mode instruction the CPU pushes the current flags and fhe **CS:EIP** registers before passing **through** an **IDT** interrupt **gate.** Some errors also generate an error code **that** identifies **the** failing segment. The **stub** code shown in Listing **1** pushes **the interrupt ID** and executes a FAR CALL task switch **to** the error handler.*

Figure 2—*The error handler's **TSS Backlink** field holds the failed task's JSS selector. **The** error handler accesses fhe stacked values using the **SS:ESP** values from that **TSS.** Note that the **stacked CS:EIP** points to the failed instruction and the **CS:EIP** in the TSS points to the instruction after the FAR CALL in fhe stub routine.*

Figure 3—The error handler displays values from the failed task's stack, dumps fields from its TSS and LDT, then halts the system. Demo Task 1 causes a variety of (deliberate!) errors based on the LPT1 DIP switches. This dump occurred after a floating-point op in a '386SX system without a numeric coprocessor. As shown on the second line, the CPU defected the error in the instruction at 000C:0000013F in Demo Task 1. Isn't this better than pinball panic or no error indication at all?

```
*** Fatal error detected...
    Int 07 at 000C:0000013F, flags 00010087, error code not used
    Coprocessor not available
       TSS Dump of [Demo Task 1] Sel=1050 Base=00130A00
        Backlink=0000 LDT Sel=1058
        CS:EIP=0030:00000561 EFLAGS=00000087 CR3=00000000
        SS:ESP=0024:00000FEC EBP=00000000 IOMapBase=0000 Trap=0000
            DS=0014 ES=0000 FS=0020 GS=001C
           EAX=00000007 EBX=0000013F ECX=00000000 EDX=00000378
           EDI=00000000 ESI=00000000
        SS:ESP 0/0000:00000000 1/0000:00000000 2/0000:00000000
       LDT Dump of [Demo Task 1] LDT Sel=1058
        0004: 00302380 00008C00
        000C: 37300178 00409810
        0014: 0020001B 00409314
        001C: 7BC00103 0040934A
        0024: 6B800FFF 0040934A
*** The system is stopped
```

Backlink field and sets the NT bit in EFLAGS.

Unlike task switches through FAR JMPs, nested tasks can return to the previous task using an IRET instruction. A normal IRET restores CS:EIP and EFLAGS from the current task's stack. If the NT bit is set, however, the CPU treats an IRET as a task switch using the TSS selector in the Backlink field. As we'll see, this lets an interrupt trigger a task switch, perform a function in complete isolation from the interrupted task, and return directly without executing any special code.

A more complex operating system than FFTS might attempt to fix up the condition causing the error and retry the failing instruction. For example, the CPU triggers Int 0B ("Segment Not Present") when an instruction uses a descriptor that is not present (P bit = 0). The error handler can reach back through the nested TSS, find the offending descriptor, make it present (perhaps by allocating a block of memory and reading a code segment from disk), then restart the failed instruction. This is obviously not for the faint of heart!

Our error handlers, on the contrary, display the values from the failed task's stack, dump fields from its TSS and LDT, and halt the system. Demo Task 1 can now cause a variety of (deliberate!) errors depending on the settings of the DIP switches on LPT1. Figure 3 shows a screen dump resulting from executing a floating-point op without a coprocessor.

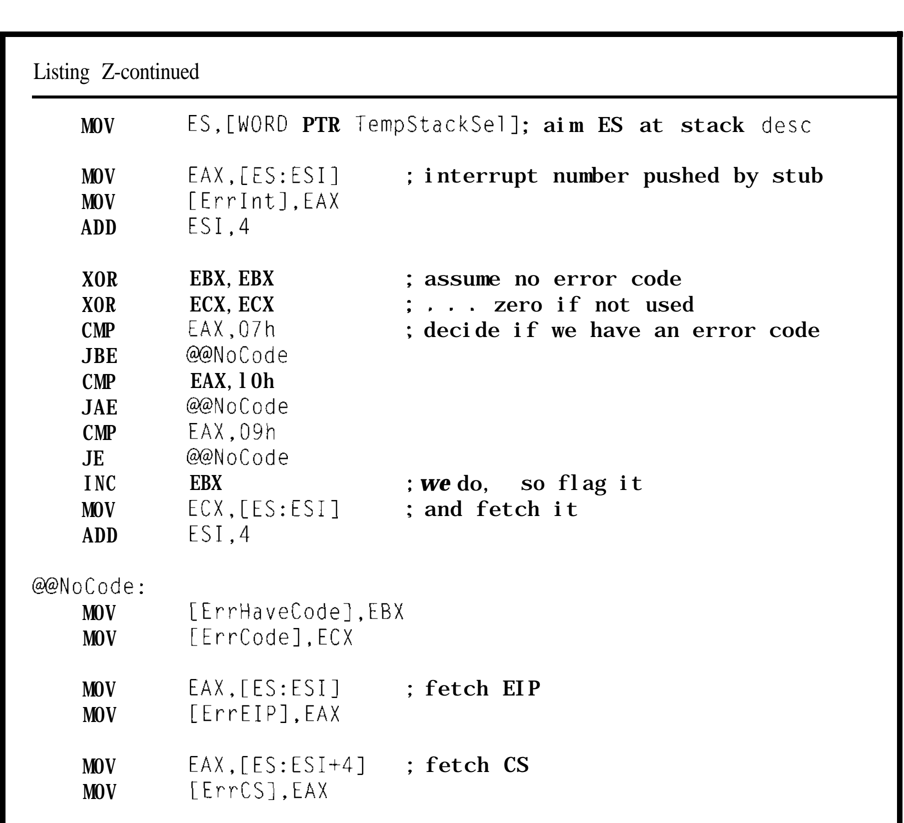
The second line in Figure 3 shows the interrupt number and the address of the instruction that caused the problem. The CS:EIP values in the TSS dump point to the FAR CALL instruction that switched into the error handler. The remainder of the registers have the same values they did when the error occurred. Reconstructing the problem is much easier when you can see what went wrong!

Fetching data from the failing task's stack is a three-step process as shown in Listing 2. First, the error handler extracts the Backlink field from its own TSS to identify the failed TSS. Next, it recovers the SS:ESP registers in use at the time of the failure and copies the corresponding stack descriptor to a temporary GDT

Listing 2—This error handler code reads the Backlink field from the error handler's TSS, locates the failed task's TSS, and copies the task's LDT stack descriptor into a temporary GDT descriptor. The error handler can then copy the task's stacked values into local variables using ES:ESI. The FFTS handler includes additional code to handle errors when the kernel's GDT stack descriptor is in use.

```
 --- fetch backlink from our TSS to the task with the error
     move ESP into ESI so we can read the stack
    LEA      EAX,[(TSS PTR 0).BackLink]
    CallSys  CGT_TASK_GETFIELD,[TaskID],EAX
    MOV      [ErrTSS],EAX

    LEA      EAX,[(TSS PTR 0).ESP]
    CallSys  CGT_TASK_GETFIELD,[ErrTSS],EAX
    MOV      ESI,EAX
    MOV      [ErrStackPtr.Off],EAX

;--- set up temp descriptor for stack
    CallSys  CGT_MEM_FINDEMPTY,GDT_TEMP_BASE,GDT_GDT_ALIAS
    MOV      [TempStackSel],EAX

    LEA      EAX,[(TSS PTR 0).SS]      ; get failing SS desc
    CallSys  CGT_TASK_GETFIELD,[ErrTSS],EAX
    MOV      [ErrStackPtr.Seg],AX

;--- copy stack descriptor from LDT to GDT
    LEA      EBX,[(TSS PTR 0).TaskLDT]
    MOVZX    EAX,[ErrStackPtr.Seg]
    AND      EAX,0FFF8h         ; convert SS descriptor to offset
    ADD      EBX,EAX            ; add to LDT base offset
    CallSys  CGT_TASK_GETFIELD,[ErrTSS],EBX
    MOV      EDX,EAX            ; save for later

    ADD      EBX,4              ; fetch second dword from LDT
    CallSys  CGT_TASK_GETFIELD,[ErrTSS],EBX

    CallSys  CGT_MEM_PUTDESC,[TempStackSel],GDT_GDT_ALIAS,\
             EDX,EAX            ; set temp descriptor to stack

;--- fetch values from that stack and sort out error codes
```

(continued)


Listing Z-continued

```
    MOV     ES,[WORD PTR TempStackSel]; aim ES at stack desc

    MOV     EAX,[ES:ESI]      ; interrupt number pushed by stub
    MOV     [ErrInt],EAX
    ADD     ESI,4

    XOR     EBX, EBX          ; assume no error code
    XOR     ECX, ECX          ; . . . zero if not used
    CMP     EAX,07h           ; decide if we have an error code
    JBE     @@NoCode
    CMP     EAX,10h
    JAE     @@NoCode
    CMP     EAX,09h
    JE      @@NoCode
    INC     EBX               ; we do, so flag it
    MOV     ECX,[ES:ESI]      ; and fetch it
    ADD     ESI,4

@@NoCode:
    MOV     [ErrHaveCode],EBX
    MOV     [ErrCode],ECX

    MOV     EAX,[ES:ESI]      ; fetch EIP
    MOV     [ErrEIP],EAX

    MOV     EAX,[ES:ESI+4]    ; fetch CS
    MOV     [ErrCS],EAX

    MOV     EAX,[ES:ESI+8]    ; fetch EFLAGS
    MOV     [ErrEFLAGS],EAX
```

entry. Finally, with ES:ESI aimed at the stack, it can copy the values into local variables.

The error handler produces the output display using the string-formatting routines in the conforming-code segment we set up last month. Those routines work with values from the caller's stack and do not affect any other system values, making them ideal for an error handler that may get control at any time.

WHEN ALL ELSE FAILS...

The error handler is a task much like the demo taskettes, except that it runs only twice: once during the initial task setup and once when an error occurs. After displaying the error information, it halts the system, effectively eliminating the need to unwind the stacks and return to the failing task.

The `TaskDispatchable` bit in the dispatching array is set when the task-initialization code creates the error-handler task. After the handler finishes preparing for the first error, it turns off its `TaskDispatchable` bit and returns to the dispatcher, leaving the context of the dispatching procedure on its stack. The only way it will regain control is through one of the stub routines after an error, not through the dispatcher's loop.

When an error occurs, the CPU restores the handler's registers from its TSS and the code finally returns from the task-dispatcher procedure. It should not call the dispatcher when it finishes handling the error because the dispatcher is not expecting a return from a task it hasn't dispatched.

The Intel System Software Writer's Guide describes a moderately complex way to integrate software- and hardware-dispatched tasks. I have not used their technique because the FFTS error handlers are quite simple. If you are building a system that must recover from errors with a bit more grace, pay attention to those suggestions!

The handler I just described can deal with all but three of the CPU's error conditions. The Intel manuals recommend that stack, double-fault,

-PRECISION FRAME GRABBER FOR ONLY $495*

— CX100 FEATURES —

- Crystal Controlled Image Accuracy
- Memory Mapped, Dual-Ported Video RAM
- Programmable Offset and Gain
- Input, Output and Overlay LUTs
- Resolution of 512x486 or Four Images of 256x243 (CCIR 512x512 & 256x256)
- Monochrome, 8 Bit, Real Time Frame Grabs
- Graphics Overlay on Live or Still Images**
- External Trigger Input
- RGB or B&W, 30 Hz Interlaced Display
- NTSC/PAL Auto Detect, Auto Switch
- VCR and Resettable Camera Compatible
- Power Down Capability
- BNC or RCA Connectors
- Built-In Software Protection**
- 63 Function C Library with Source Code
- Text & Graphic Library with Source Code
- Windows DLL, Examples and Utilities
- Software also available free on our BBS
- Image File Formats: GIF, TIFF, BMP, PIC, PCX,TGA and WPG

Introducing the CX100 precision video frame grabber for OEM, industrial and scientific applications. With sampling jitter of only ±3 nS and video noise less than one LSB, ImageNation breaks new ground in imaging price/performance. The CX100 is a rugged, low power, ISA board featuring rock solid, crystal controlled timing and all digital video synchronization. Software developers will appreciate the simple software interface, extensive C library and clear documentation. The CX100 is a software compatible, drop-in replacement for our very popular Cortex I frame grabber. Call today for complete specifications and volume pricing.

ImageNation Corporation
Vision Requires Imagination
800-366-9131

** THESE OPTIONS AVAILABLE AT EXTRA COST.
* $495 IS DOMESTIC, OEM SINGLE UNIT PRICE.

P.O. BOX 276 BEAVERTON, OR 97075 USA PHONE (503) 641-7408 FAX (503) 643-2458 BBS (503) 626-7763

#121

and invalid TSS error handlers use IDT task gates rather than interrupt or trap gates. In each of these cases, the currently active stack may not be valid or may not have enough room for the error handler's use. Any attempt to push data onto a bad stack causes further errors and may force the CPU into shutdown.

A task gate is Yet Another Descriptor that specifies a TSS selector in place of the usual code-segment selector. When an error occurs, the CPU uses the corresponding IDT task gate to switch tasks without pushing any information on the failed task's stack, thus ensuring no further errors occur. The error handler's Backlink field points to the failed task and the handler may return using an IRET after resolving the problem.

If the error condition produces an error code, as is true for these three errors, the CPU pushes it onto the error handler's stack. Because the task switch occurs at the failing instruction, the TSS fields contain all of the information required to locate the problem. There is no need to find the failed task's stack and exhume values from it.

Tasks activated by an IDT task gate cannot use the FFTS task dispatcher because the CPU plops the error code atop the stack contents defined by the SS:ESP fields in the handler's TSS. This disturbs the previous return context and results in a protection error when the CPU attempts to resume execution with a "bad" stack. Not a pretty sight.

I defined three separate tasks for Int 08, Int OA, and Int OC that expect to find an error code on their stacks. The main error handler installs these three task gates after preparing the rest of the IDT interrupt gates. The task dispatcher resets the TaskDispatchable bit for these tasks when it creates them, thus preventing any execution except when an error occurs.

Because a task gets control immediately, you cannot aim multiple task gates at the same TSS if you must know which interrupt caused the switch. That's why FFTS has four error handlers: three separate tasks for the three errors that may have corrupt stacks and one task for the remaining 253 cases with stub routines to save the interrupt ID. You may prefer a separate task for each of the CPU error conditions, plus one more for all the other cases that "can't happen here."

Once again, remember that writing comprehensive error handlers is exceedingly difficult. The code I've described and implemented is barely the beginning of a real operating system's features. Even though FFTS needs additional ruffles and flourishes, I plan to favor simplicity over capability. Download the code and spend a while thinking it over-you're sure to find ways to improve it!

If you think all this is too complex for words, compare Figure 3 with the results of a similar goof in real mode. Maybe this protected-mode stuff is worthwhile?

Listing 3—The system task-switches to this interrupt handler whenever an interrupt occurs on either IRQ 5 or IRQ 7. Timer 0 on the Firmware Development Board produces a 1-ms square wave on IRQ 5. The 8259 interrupt controller produces a default IRQ 7 interrupt when the IRQ 5 input goes low during the CPU's interrupt acknowledge sequence. The two cases are distinguishable by reading the 8259's In-Service Register. The handler must not send an EOI to the 8259 when a default IRQ 7 occurs.

```
     UseTaskCS

     PROC TaskProcInt

@@Again:
     MOV    EDX,SYNC_ADDR
     IN     AL,DX              ; raise the blip
     OR     AL,40h
     OUT    DX,AL

     MOV    AL,00001011b       ; OCW3 with read ISR set
     OUT    I8259A,AL          ; tell the 8259
     IN     AL,I8259A          ; read the ISR
     TEST   AL,00100000b       ; is IRQ 5 active?
     JNZ    @@Normal           ; yes, so do a normal interrupt

     MOV    EDX,SYNC_ADDR;     no, mark a default interrupt
     IN     AL,DX
     OR     AL,20h
     OUT    DX,AL
     AND    AL,NOT 20h
     OUT    DX,AL

     INC    [Int7Counter]      ; no, we have a default IRQ 7
     JMP    @@Done             ; do not send EOI for this one

@@Normal:
     INC    [Int5Counter]      ; record a normal interrupt
     MOV    AL,NS_EOI          ; send EOI to controller
     OUT    I8259A,AL

@Done:
     MOV    EDX,SYNC_ADDR
     IN     AL,DX              ; lower the blip
     AND    AL,NOT 40h
     OUT    DX,AL

     IRET                      ; return to previous task

     JMP    @@Again            ; and repeat!

     ENDP TaskProcInt

     EndTaskCS
```

TICKING A TASK

In *INK 50,* we found that a 33-MHz '386SX responds to an external

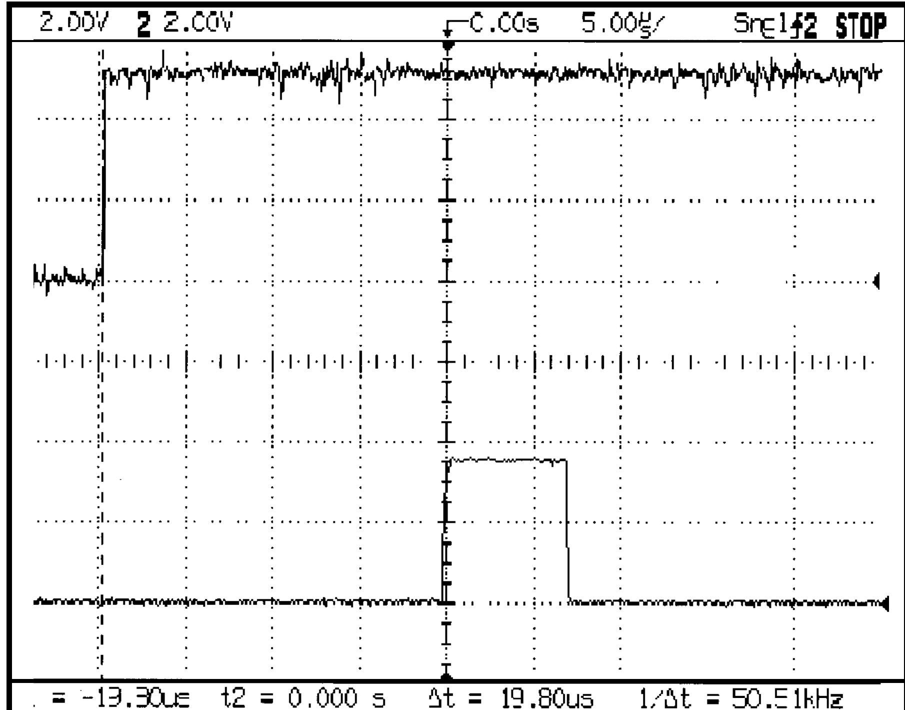


Photo 1—*The response to an external interrupt can be rather slow when the interrupt handler is a separate task invoked through an IDT task gate. FDB Timer 0 generates IRQ 5 on the top trace and the interrupt handler produces the blip on the lower trace. The 19.8-µs (680 cycles at 33 MHz) response time shown here is roughly 2.5 times longer than the delay through an interrupt gate.*

interrupt in about 7 µs when the handler uses an IDT interrupt gate. Now that we can set up and use separate tasks, it's reasonable to ask what the response time for a complete context switch might be. The CPU is obviously performing more work on our behalf while switching from one task to another. So, how long does it take?

I modified Demo Task 2 to set up the machinery required to produce an interrupt and then display the results on the VGA. This gives you a real-time view of what's happening down at the grubby hardware level. Because we've used protected-mode interrupts before, I'll skip the detailed listings and cover the new stuff.

Timer 0 in the Firmware Development Board's 82C54 chip produces a 1-ms square wave on the IRQ 5 ISA bus line. The demo tasks require several milliseconds to update the VGA and LCD display and thus allow several timer interrupts while they are executing. If you don't have an FDB in your system, you can modify the code to use the system-board timer.

I remapped the system's two 8259 interrupt controllers (or, more precisely, the LSI slivers that emulate 8259s) to produce Int 50-57 and Int 70-77 (hex), respectively. Because we are interested only in IRQ 5 on Int 55, I cleared just one bit in the primary controller's Interrupt Mask Register. All other external interrupts remain masked off.

The interrupt handler shown in Listing 3 is a separate task that cannot use the normal FFTS dispatcher procedure. A task gate, much like the gates used for the CPU's error handlers, contains the IRQ 5 handler's TSS selector. When an IRQ 5 interrupt occurs, the CPU reads the interrupt number from the 8259, locates the task gate in the IDT, and switches to the handler task.

After all the setup is complete, the Demo Task 2 code executes an **ST I** instruction to set the CPU's IF and enable external interrupts. Up to this point, the FFTS kernel has been an external-interrupt-free zone.

Photo 1 shows the results. The rising edge on IRQ 5 in the top trace triggers the interrupt. About 20 us later, the second trace rises to show that the interrupt handler is in control. The '386SX CPU runs at 33 MHz, so you are looking at about 650 clock cycles of delay. A few microseconds vanish while producing the output pulse, but this is about as good as it gets.

Dig out your back issues. Photo 1 in ***INK 50*** shows a 7-µs response through an interrupt gate (the caption's "7 ms" is a typo). Photo 1 in ***INK 54*** shows that a task switch requires about 15 us. It shouldn't be surprising that an interrupt plus a task switch requires somewhat more time than a task switch alone, but less than both together.

Protected mode offers a variety of ways to respond to interrupts. YOU can use an interrupt gate for handlers that perform relatively simple actions or task gates that switch the entire CPU context. You may also, of course, perform your own task switch in firmware at the risk of taking more time to accomplish less while evading the CPU's hardware protection. Unlike running code in real mode, you've got choices for your handlers.

SWITCHED SUPPRESSION

During each task switch, the CPU reloads all of its registers from a TSS. Although we haven't covered all the implications yet, that means the EFLAGS register is unique to each task. Bit 9 of EFLAGS is more commonly known as IF (Interrupt Flag).

Get it?

The IRQ 5 handler produces the upper trace of Photo 2. Although Timer 0 runs continuously, interrupts are enabled only when Demo Task 2 is active, as shown in the lower trace. External interrupts occurring while IF is zero are not recognized, just as in real mode.

Moral of the story: in a multitasking system, you must enable interrupts in every task if you want consistent response times. If any task disables interrupts, you will get gaps while interrupts receive no attention at all.

Because user tasks should not have that much influence over the system's operation, the two-bit I/O Privilege Level in EFLAGS affects the

STI and CLI instructions. If the current task is less privileged than the IOPL setting, the CPU invokes the general-protection handler. This is an effective way to prevent Level 3 user tasks from clobbering the whole system.

Normally, you change EFLAGS by pushing it onto the stack, popping it into EAX, altering a few bits, pushing EAX onto the stack, and popping the new value back into EFLAGS. In protected mode, the CPU will not change the IOPL field unless the task is already running at Level 0, thus preventing user tasks from changing their own IOPL and gaining access to sensitive system resources.

There are other complications that we'll explore in due time. For now, just remember that interrupts are no longer a private thing.

CAP'N QUIRK TO THE BRIDGE!

You must cultivate the ability to read hardware data sheets completely and accurately if you intend to write good firmware. It also helps if you can read between the lines, because that's where the quirks are hidden. Consider this excerpt from the Intel 8259 data sheet:

In both the edge- and level-triggered modes, the IR inputs must remain high until after the falling edge of the first INTA. If the IR input goes low before this time, a DEFAULT IR7 will occur when the CPU acknowledges the interrupt.

Novices skip over this stuff because it doesn't make much sense. An engineer with more experience sticks a red Post-It note on the page and scrawls timing diagrams in the margin. The Perfect Master perceives the implications without further effort.

Me, I just sort of muddle along.

The Original PC used edge-triggered interrupts, creating compatibility barnacles that force all ISA bus systems into the same mode. EISA systems may (and Micro Channel systems always) use level-triggered interrupts with cards built to share interrupts. You can actually use level-triggered interrupts in an ISA bus system, although I'll leave that as an exercise for you.

There are three requirements for a valid edge-triggered interrupt: the IRQ line must have a rising edge, it must remain high until the CPU acknowledges the interrupt, and (obviously) it must go low to prepare for the next interrupt. The 8259 holds its INT output high whenever it has an interrupt pending.

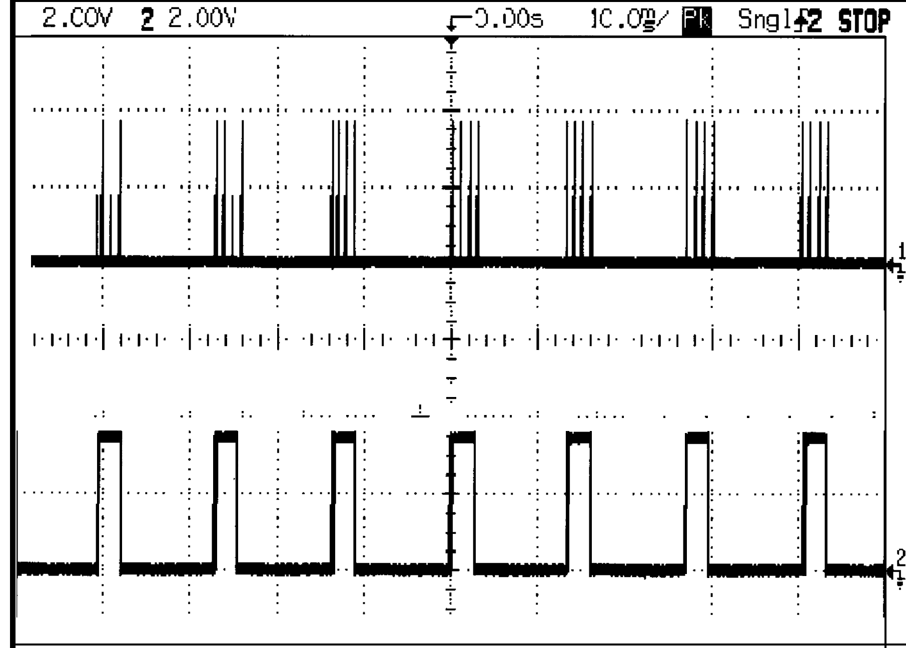


Photo 2—*Interrupts are active only when the CPU's Interrupt Flag is set. Demo Task 2 executes an* STI *instruction after preparing for interrupts, resulting in a system that responds to interrupts only when that task is active. The IRQ 5 handler task produces the clusters of pulses in the top trace. Demo Task 2 produces the blips in the lower trace when it is active.*

Contrary to popular assumption, however, the 8259 does not "remember" an interrupt that Goes Away before the CPU detects it, even in edge-triggered mode. If the IRQ input goes low, the 8259 lowers its INT output. The CPU will not detect an interrupt.

The data sheet description applies only to IRQ inputs that Go Away in the short interval when the 8259 is processing the CPU's first INTA pulse. In that situation, the CPU detects a pending interrupt, starts an interrupt-acknowledgment cycle, and then suddenly discovers that it doesn't have a valid IRQ input. What to do?

You could argue that the 8259 should issue an interrupt for the now-vanished IRQ. However, that could cause system problems if the interrupting hardware no longer needs service. Worse, the interrupt could have been a brief glitch on the line rather than a valid signal.

What the 8259 actually does is generate an IRQ 7 interrupt with ISR bit 7 set to zero. Thus, the IRQ 7 interrupt handler must distinguish between valid IRQ 7 hardware interrupts and default interrupts. Invalid timing on any interrupt line, including IRQ 7, causes a default IRQ 7.

When the interrupt handler detects a default IRQ 7 event, it must not send an EOI command to the 8259. An EOI resets the highest-priority ISR bit and may discard a valid interrupt if the 8259 recognizes a new IRQ signal while the CPU is busy with the default IRQ 7.

Because Timer 0 is not synchronized with the CPU's clock, we can be sure that it will eventually violate the (unspecified) timing specs causing a default IRQ 7 interrupt. All we have to do is sit back and watch.. .

Demo Task 2 installs task gates at both Int 55 and Int 57 to invoke the handler task in Listing 3. The handler determines which interrupt invoked it by reading the 8259's ISR. If bit 5 is

clear, meaning that a valid IRQ 5 did not occur, then the interrupt must be a default IRQ 7. The code pulses parallel port bits 6 and 5 to give us a real-time picture of what's happening.

The main loop of Demo Task 2 displays the two interrupt counters and their ratio, scaled by a factor of one million, on the VGA display each time it runs. The interrupt handler task simply increments the counters and returns because we don't have nearly enough time between interrupts to update the screen.

Photo 3 catches a default IRQ 7 in action. The bottom trace goes high when Demo Task 2 is active. External interrupts are enabled a few microseconds before the rising edge of that pulse when the CPU exits from the task switch instruction. The Timer 0 pulse on IRQ 5 shown in the top trace falls just before that key event.

The blip on Trace 2 marks a default IRQ 7 interrupt. Trace 3 shows two interrupt handler task activations: first for the default interrupt and then as a valid IRQ 5 after the rising edge of Timer 0.

After about 64 hours of continuous execution, the program recorded 68.3 million IRQ 5 and 11,815 default IRQ 7 interrupts. That works out to 172 parts per million-infrequent enough that you'd never see one if you weren't looking directly at it.

Don't get too nervous about this condition, though. It only occurs when the interrupt source Goes Away precisely when the CPU is acknowledging the interrupt. If your interrupts are enabled all the time and the pulse stays high longer than the maximum CPU response time, you'll never see a default IRQ 7.

In any case, build a test into your IRQ 7 and IRQ 15 handlers just in case you get a glitch. Always accumulate a counter, then examine it once in a while. Who knows? You might see a one part-per-million blip occasionally!

RELEASE NOTES

The demo taskettes include test code for the error and interrupt handlers. Demo Task 1 monitors LPT1 and triggers a variety of (deliberate!) errors to verify that the handler tasks work correctly. Demo Task 2 installs an interrupt handler task, activates Timer 0 on the Firmware Development Board, counts the number of IRQ 5 and IRQ 7 interrupts, and displays running totals on the system's VGA. Demo Task 3 simply ticks a count on the VGA and graphic LCD panel.

Next month, we'll fire up the system board's real-time clock interrupts, twiddle a watchdog, read a serial number, put some characters on the FDB's character LCD, and look at memory allocation. ❑

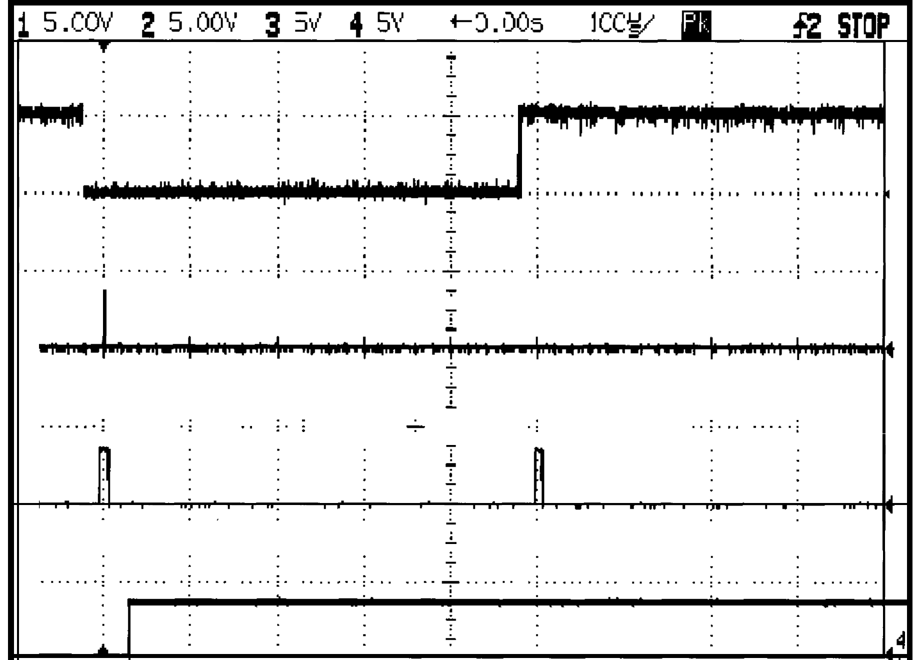


Photo 3—The 8259 interrupt controller generates a default IRQ 7 if an *interrupt request input becomes inactive during the CPU's hardware response. The falling edge of IRQ 5 in Trace 1 triggers the* default *IRQ* 7 shown in *Trace 2 because if occurs* just *as the CPU becomes enabled for* interrupts *in Demo Task* 2. Trace 3 goes high *when the* system responds to either *IRQ* 5 or *IRQ interrupts.* Trace 4 shows the start of Demo *Task* 2. *The* interrupt handler counts *IRQ* 5 *and IRQ 7 events. In this system, there are about 170 IRQ 7 interrupts per million IRQ 5 inferrupfs.*

Acronyms

CPL	Current Privilege Level
DPL	Descriptor Privilege Level
EOI	End Of Interrupt (command)
FDB	Firmware Development Board
FFTS	Firmware Furnace Task Switcher
GDT	Global Descriptor Table
GDTR	GDT Register
IDT	Interrupt Descriptor Table
IF	Interrupt Flag
IOPL	I/O Privilege Level
LDT	Local Descriptor Table
LDTR	LDT Register
NT	Nested Task
P bit	Present Bit (in a PM descriptor)
RF	Resume Flag
RPL	Requestor Privilege Level
TF	Trap Flag
TR	Task Register
TSS	Task State Segment

Ed Nisley, as Nisley Micro Engineering, makes small computers do amazing things. He's also a member of Circuit Cellar INK's engineering staff. You may reach him at *ed.nisley@circellar.com.* or *74065.1363@compuserve.com.*

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.

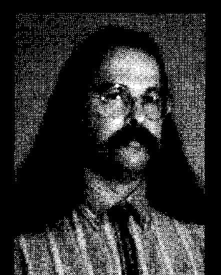
IRS

425 Very Useful
426 Moderately Useful
427 Not Useful

FROM THE BENCH

Jeff Bachiochi

Vaporwear: Revealing Your Humidity



Sultry summer nights, stuck doors—the genre of humidity. But, humidity is measured just about everywhere. Jeff covers hygrometer basics and runs us through the fundamentals of how to calibrate our own humidity sensor.

You've probably heard someone comment about forecasting the weather from an ache or pain. What they are actually feeling is the change in humidity as their bones and joints swell or shrink from a change in the air's moisture content. We live somewhere between the extremes of a desert's lack of moisture and a sauna's abundance of it.

Take those muggy summer days (please); they can be brutal. The air seems so heavy. And, for good reason-it actually is. Humid air is saturated with water vapor. In this gaseous state, the water heeds the same rules as the other gases which combine to make air.

The relationship between air's pressure, volume, and temperature are defined in the Ideal Gas Law:

$$\frac{PV}{T} = K$$

where P is pressure, V is volume, T is temperature, K is the gas constant times the number of moles of gas. In other words, pressure and volume are inversely proportional, whereas temperature is proportional to both pressure and volume.

Water is in a significant part of our lives. It's in our bodies, what we eat, and the air we breathe. The moisture content of air can be measured by weighing all the water-vapor molecules with respect to other gas molecules-not an easy task for tweezers, a magnifying glass, and a postage scale.

However, it can be calculated from knowing the dew/frost point (DFP) of the air. The DFP temperature represents the temperature that the water in the air becomes saturated and condenses into water or ice. The warm, moist air we exhale on a cool morning is chilled to dew point and instantly condenses into water droplets or fog. Measuring the exact temperature at which the condensation takes place lets the relative humidity (RH) be calculated.

Humidity affects us on a personal level within our own comfort zone. It is important to note here that humidity is just as important to other activities that operate in severe environments. Industrial furnaces or upper atmospheric experiments pose special problems to the measurement of humidity and require specially designed sensors and/or sampling equipment.

But, let's try to remain within our comfort zone here for the remainder of this discussion.

COMFORT ZONE MEASUREMENTS

For most of us, while the outside temperature varies within a range of 0–100°F, our artificial living environment stays within 65–75°F. Those of us with base-board heat don't have much control over humidity. We might keep a kettle of water on the wood stove or run a humidifier to keep a bit of humidity in the air, but in general most of us don't have a hygrometer on the wall next to the thermostat.

Before we added on to our cottage here in New England, it was heated by a hot-air system. This old system, antiquated as it was, did have a humidistat located within the central air duct. Whenever hot air was moving through the duct, a fine mist of water vapor was introduced in an attempt to control the humidity. I never actually felt the effects of the humidistat

Sensor type	Accuracy	Operational temperature (°C)	Operational life	Avoid	Output	Response time	Response step	Operational range	Cost
Resistive	high	0 to 100	1% year drift	ammonia	impedance log (%RH)	<60	30–80%	30–90%	low
Capacitive	high	–20 to 40	1% year drift	acetone	capacitance linear (%RH)	<60	10–90%	5–95%	low
Saturated salt	medium	–10 to 50	less than 5 years	unpowered in high RH	temperature D/F point	<5 min	30–50°C	12–100%	medium
Chilled mirror	high	0 to 40	less than 10 years	high salt	temperature D/F point	<60	–20–0°C	–40–50°C	high
Electrolytic	medium	0 to 40	500K PPM hours	>1000 PPM	volts linear (PPM)	<30 min	0–100 PPM	5–1000 PPM	medium

Table 1—*There are several different kinds of humidity measuring systems, each with its own characteristics. Which you use depends on the application.*

because it had become clogged with iron from our water long before I moved in. In fact, I didn't know it existed until I removed it from the ductwork being thrown out by the contractor who was installing a new zone-controlled hot-water system.

You've no doubt noticed the effects of low and high humidity in your own home. Low humidity dries our throats and mouths while high humidity makes our skin feel clammy—the perspiration has a hard time evaporating. Air conditioning's major activity of cooling the air also has the secondary effect of removing humidity. It's probably true that this lowering of humidity makes us most comfortable.

Warmer air holds more water vapor than cooler air since air molecules move further apart as the temperature rises. At any stable, ambient temperature, when the air becomes unable to accept more water-vapor molecules, it reaches its saturation point, which is otherwise known as 100% relative humidity (other humidity definitions are listed in the sidebar). Remember though, relative humidity of air, which hinges on the saturation point, changes with temperature. (Actually, the percentage of saturation and the percentage of humidity are not synonymous. However, within the comfort zone, the terms can be used interchangeably.)

HUMIDITY SENSORS

Since RH, DFP, and parts-per-million (PPM) measurements are all directly related, the humidity sensor you choose will only measure one of those parameters. Your choice therefore depends on the accuracy, response time, operating range, and of course budget that you need.

RELATIVE HUMIDITY SENSORS

Direct measurement of relative humidity can be achieved through resistive, capacitive, or liquid sorption sensors. The resistive sensor uses a water-vapor-permeable coating over twin electrodes laid out in an alternating pattern on a ceramic substrate. An alternating current excites the sensor. The sensor's impedance is affected when the resin's ions are made mobile by the penetration of water vapor. Impedance changes are on the order of 2 kΩ to 10 MΩ.

Capacitance sensors can be manufactured similarly to the resistive sensor or, as the more traditional capacitor, with parallel plates. In either case, the polymer dielectric material absorbs and desorbs water vapor, affecting the total capacitance of the sensor. Capacitance change is usually about 30%.

The sorption sensor changes volume by absorbing and desorbing water vapor. This change in volume must be measured by a strain gauge, pressure sensor, or other method.

DEW/FROST POINT SENSORS

DFP measurements are realized through either a chilled mirror or saturated-salt sensor. A mirror placed in the path of an optical transmitter-receiver pair scatters the reflection when water vapor condenses on its surface. The mirror's temperature is adjusted to remain at the condensation point and the temperature of the mirror's substrate is reported as the DFP.

Electrically conductive lithium chloride can be used to measure the DFP by applying it to an absorbent fabric. The fabric absorbs and desorbs water vapor. Absorbing water vapor makes the fabric conductive so that a current flows through it and a small heating coil. The warmed fabric causes more water to evaporate, which reduces the current passing through it. As an equilibrium is reached between the current flow and heat generated, the measured temperature is the DFP.

PARTS PER MILLION SENSORS

The third humidity unit of measurement is PPM of water vapor present within a gas. This might be measured by electrolytic or piezo-

Humidity: the amount of water vapor present in a gas in a pure (e.g., nitrogen) or mixed form (e.g., air).

Relative Humidity (RH): the ratio of the water vapor's mass to the dry gas's mass at a given temperature.

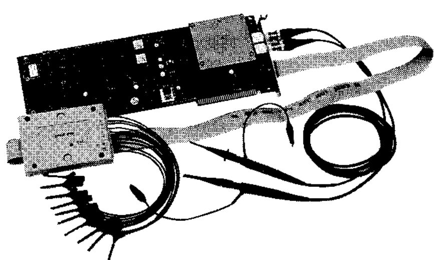
Absolute Humidity: the water vapor's mass per unit volume of dry gas at a given temperature.

Dew or Frost Point (DFP): the temperature at which a volume of gas that is being cooled can no longer support the water vapor present. It is known as the dew point if it condenses into water and as the frost point if it condenses into ice.

Parts Per Million (PPM): a million times the molecular weight of water present compared to either the dry gas's weight (PPMw) or volume (PPMv).

PC-Based Instruments
200 MSa/s DIGITAL OSCILLOSCOPE

HUGE BUFFER
FAST SAMPLING
SCOPE AND LOGIC ANALYZER
C LIBRARY W/SOURCE AVAILABLE
POWERFUL FRONT PANEL SOFTWARE



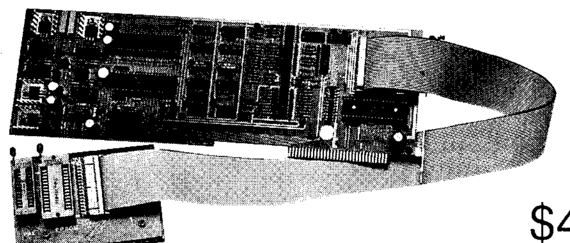
DSO Channels
2 Ch. up to 100 MSa/s
or
1 Ch. at 200 MSa/s
4K or 64K Samples/Ch
Cross Trigger with LA
125 MHz Bandwidth

Logic Analyzer Channels
8 Ch. up to 100 MHz
4K or 64K Samples/Ch
Cross Trigger with DSO

$1799 - DSO-28204 (4K)
$2285 - DSO-28264 (64K)

Universal Device Programmer

PAL
GAL
EPROM
EEPROM
FLASH
MICRO
PIC
etc..

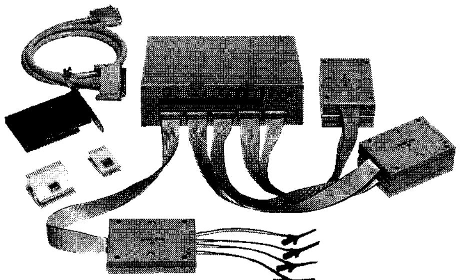


$475

Free software updates on BBS
Powerful menu driven software

400 MHz Logic Analyzer

- up to 128 Channels
- up to 400 MHz
- up to 16K Samples/Channel
- Variable Threshold Levels
- 8 External Clocks
- 16 Level Triggering
- Pattern Generator Option



$799 - **LA12100 (100 MHz, 24 Ch)**
$1299 - **LA32200 (200 MHz, 32 Ch)**
$1899 - **LA32400 (400 MHz, 32 Ch)**
$2750 - **LA64400 (400 MHz, 64 Ch)**

Call (201) 808-8990

Link Instruments
369 Passaic Ave, Suite 100, Fairfield, NJ 07004 fax: 808-8786

#125

resonance sensors. The electrolytic sensor uses a DC voltage applied to dual electrode windings along the inside of a tube. A phosphorous pentoxide coating absorbs water vapor from the gas as it travels through the tube. Electrolysis breaks the water down into oxygen and hydrogen at the rate of one molecule of water for every two electrons. Current flow represents removed (counted) water molecules. This value along with the temperature and flow rate yields the PPM by volume.

The piezoresonance sensor is crystal coated with a humidity-sensitive material. The crystal's vibrating frequency is directly affected by a change in its mass as the coating absorbs water from and desorbs water to the air.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

Many industrial processes require highly accurate and stable humidity control. NIST provides a calibration standard for the industry of down to 0.1°C (DFP). Instruments traceable to this standard transfer calibration to other devices. The necessity for keeping transfer standards in continuous calibration is costly since manufacturers of most sensor systems need to recalibrate the system whenever a sensor is replaced. This can be simplified somewhat when the sensor manufacturer provides sensor linearity data via EPROM. Then, the user can replace the system's firmware and sensor in matched pairs.

Such accuracy isn't always necessary. I want to measure the RH—the normal atmospheric and temperature conditions—of my surroundings. Most companies I've talked with don't want to deal with the little guy. This is a continual problem for those of us who want to experiment a bit without committing to 100k pieces over the next year. These companies are in business to sell complete systems based on small sensor technology. Many sensors carry with them a hefty price tag and rightly so. The interface electronics necessary to produce accuracy and traceabilty don't come cheap.

Still there are some firms willing to make their sensor technology available for a price, although it is by no means small. Table 1 offers a sample of available humidity sensor characteristics.

IT'S ALL RELATIVE

Although I could have chosen a number of different sensors, I will be using Panametrics' Humidicap-2. This sensor is one of the smallest available. It comes in a TO18-type can with the top open to the atmosphere. A small plastic sleeve prevents even the clumsiest enthusiast from harming the delicate wire bonds to the sensor.

Although physically delicate, the sensor is rated to operate from -40 to +50°C with negligible temperature dependence above freezing. Bulk capacitance at 33% RH is 207 pF±31 pF (15%). Capacitance change from 10 to 90% RH is typically 12% of bulk. The linearity is ± 1% over that range, which means no algorithm is necessary to correct for nonlinearities.

Figure 1 shows a basic circuit for converting capacitance to pulse width and voltage. The actual board is pictured in Photo 1. Here, a MAX7556 (a low-voltage version of the dual 555 timer) is used. The first stage is connected to form a constant-frequency pulse generator. The second stage, triggered from the first, creates a varying pulse width proportional to its RC time constant.

Since the resistance is fixed, the change in capacitance is directly proportional to the pulse width. This pulse width could be measured digitally through a microprocessor's timer input and converted to the corresponding humidity level. Alternatively, the PWM signal can be fed into a low-pass filter and measured as a voltage.

I used an additional dual op-amp for an offset stage to adjust 0% RH to 0 V and a gain stage to allow 100% RH to be measured as 5 V. The O-5-V signal can be used directly by most A/D converters. The op-amp needs a bit of head room on the power supply, so I used a MAX680 to produce ±9 V from the +5-V circuit input.

Calibration techniques usually call for special salt solutions to create accurate humidity levels in closed containers. The sensor is inserted into the chambers and allowed to stabilize. Each salt solution maintains a particular humidity level. The circuit measurements taken in two humidity environments indicate the slope of the sensor's output in relation to the humidity level.

To reduce the calibration costs to a reasonable level, I was prepared to create my own humidity chamber once I had a way to measure it. On a trip to the local hardware store, I browsed the thermometer section. With a couple of thermometers, I could rig up a Sling Psychrometer and measure wet- versus dry-bulb temperature differences and thus relative humidity. Then, I noticed the combination thermometer/hygrometers. They ranged in price from $4.99 to $32.95. After comparing the humidity readings and display scales, I found the least expensive and most expensive models to be comparable.

My humidity chamber consists of the upstairs bathroom with a portable humidifier. Prior to taking the first reading I let the sensor, circuitry, hygrometer, and humidifier stand for an hour to let all the apparatus get climatized to the present environment. An initial measurement shows 0.136 V for a humidity level of 15%. After three hours with the humidifier on, a second measurement shows 0.147 V for a humidity level of 70%.

Obviously, the voltage readings can be used to calculate a percentage of humidity. Assuming linearity, we can calculate the sensor's output for the extremes of 0% RH and 100% RH. To do this, we first of all have to find the slope of the line between the initial and subsequent voltage readings:

$$\Delta = \frac{V_{sub} - V_{init}}{RH_{sub} - RH_{init}} = \frac{0.147\,V - 0.136\,V}{70\% - 15\%} = \frac{0.2\,\mu V}{RH\%}$$

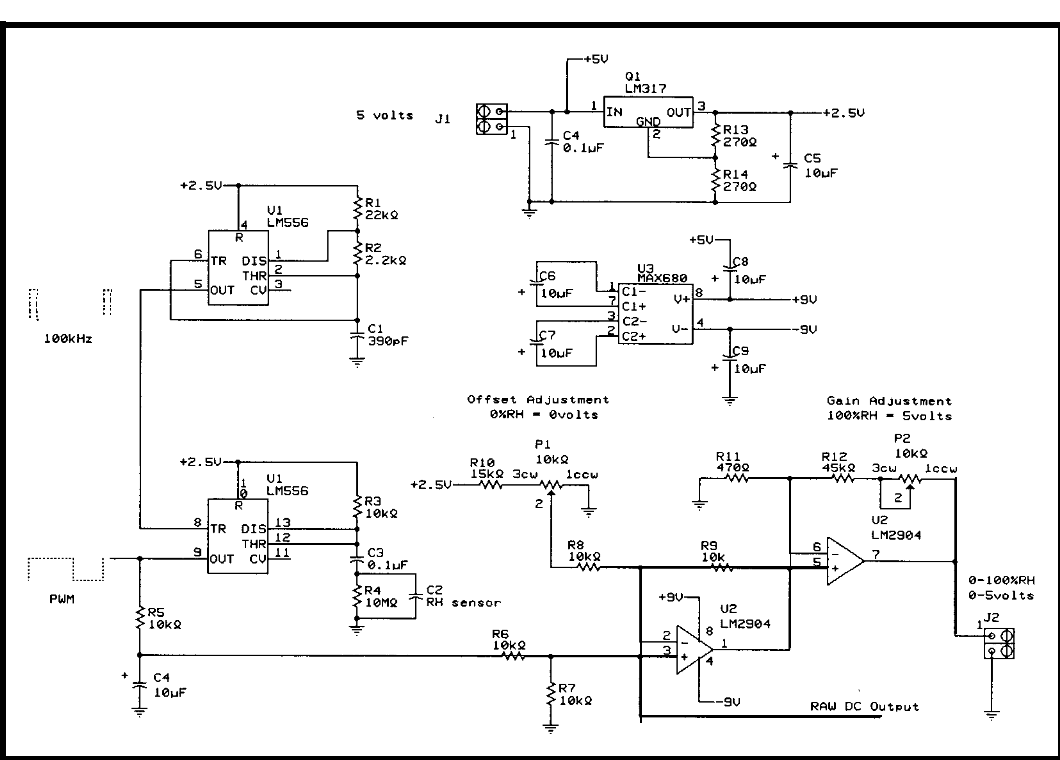
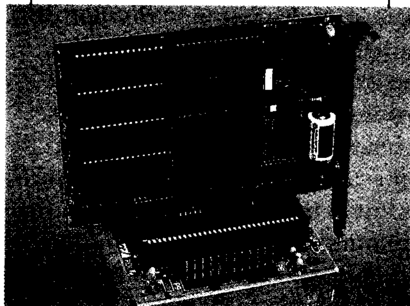
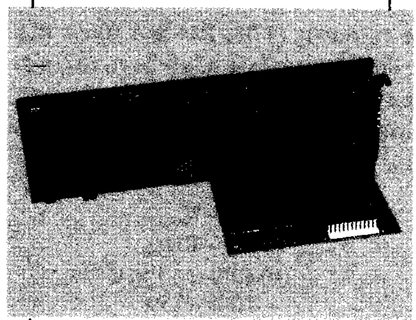


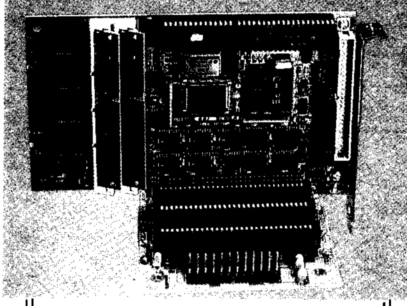
Figure 1-A *stable oscillator triggers a one-shot circuit where* **PWM** *is proportional* **to** *the humidity sensor,* **C2.** *A voltage from* **the filtered** *PWM is offset and multiplied to approximate a C-5-V output equivalent to* **0–100%** *relative humidity.*



SOLID STATE DISK — $135"
½ Card 2 Disk Emulator
EPROM, FLASH and/or SRAM
Program/Erase FLASH On-Board
1M Total, Either Drive Bootable



25MHZ 386DX CPU — $695*
Compact AT/Bus or Stand Alone
On-Board SVGA, IDE, FDC, 2 Ser/Bi-Par
FLASH/SRAM Drives to 2.5M
Cache to 128K, DRAM to 48M



TURBO XT w/FLASH DISK — $266*
To 2 FLASH Drives, 1M Total
DRAM to 2M
Pgm/Erase FLASH On-Board
CMOS Surface Mount, 4.2" x 6.7"
2 Ser/1 Par, Watchdog Timer

All Tempustech VMAX products are PC Bus Compatible. Made in the U.S.A., 30 Day Money Back Guarantee
*QTY 1, Qty breaks start at 5 pieces.

TEMPUSTECH, INC.
TEL: (800) 634-0701
FAX: (813) 643-4981

Fax for fast response! 295 Airport Road Naples, FL 33942

#126

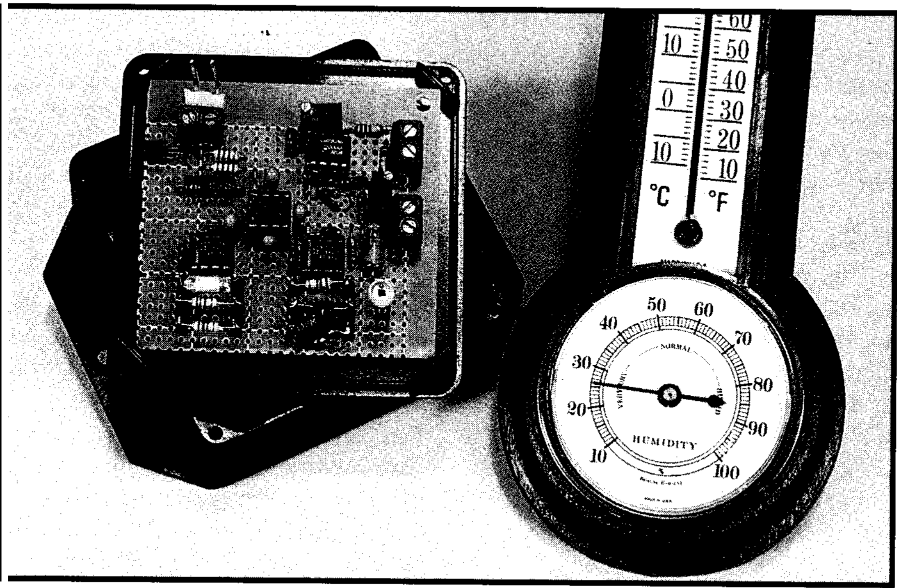


Photo 1—The *prototyped circuit is mounted on perf board for easy experimentation. The Humidicap sensor* ***with*** *this* ***circuitry*** *enables humidify* ***to*** *be read as pulse width or* ***voltage.***

With the slope of the line, we can now find the voltage at 0% RH:

$$V\ @\ 0\%\ RH = V_{init} - \Delta V \times RH_{init}$$
$$= 0.136\ V - 0.0002\ V \times 15\%$$
$$= 0.133\ V$$

and the voltage at 100% RH:

$$V\ @\ 100\%\ RH = V_{sub} + \Delta V \times 100 - RH_{end}$$
$$= 0.147\ V + 0.0002\ V \times (100-70)$$
$$= 0.153\ V$$

Knowing the voltage readings at 0 and 100% RH, we are able to set the op-amp's offset and gain. For the offset, apply the voltage calculated at 0% RH to the input of the op-amp and adjust the offset to 0 V out. Similarly, for the gain, apply the voltage calculated at 100% RH to the input and adjust the gain to 5 V out. Since these adjustments interact, they should be done more than once.

Even though an 8-bit ADC may not seem like overkill in this case, the O-5-V input converts the percentage of RH at about 0.2% RH per bit, an amount which no one will even notice. ❑

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

SOURCES

Relative humidity products:
General Eastern Instruments
High Voltage Engineering Division
20 Commerce Way
Woburn, MA 01801
(617) 938-7070

Phys-Chem Scientific Corp.
36 West 20th St.
New York, NY 10011
(212) 924-2070
Fax: (212) 243-7352

Panametrics
221 Crescent St.
Waltham, MA 02154-3497
(800) 833-9438

Humidity transmitters and moisture analyzers:
EG&G Environmental Equipment
217 Middlesex Tpk.
Burlington, MA 01803
(617) 270-9100

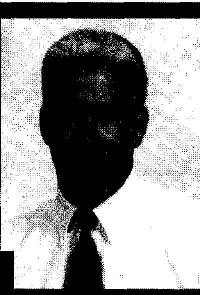
Humidity, temperature, barometric pressure instruments:
Rotronic Instrument Corp.
7 High St., Ste. 207
Huntington, NY 11743
(516) 427-3994

IRS

428 Very Useful
429 Moderately Useful
430 Not Useful

A Saab Story

A Tale of Speed and Acceleration



Tired of being tormented for being a SAAB fanatic, Tom decides to come up with a "no lie" comparison of speed and handling tips and tricks. His weapon: the Silicon Microstructures 7130-002 accelerometer.

SILICON UPDATE

Tom Cantrell

I once read a review in a car magazine that opened with "You know those Saab owners, the ones who go to foreign movies and build airplanes in their basement.. ." Ho, ho, ho.

The pundits use the same "Saab Story" title as an oh-so-clever way to get in a few digs-perhaps a story about a breakdown in the boonies and an encounter with a grizzled pump jockey. Claims he can fix "them furren jobs" are rendered suspect by his struggle with the hood (it flips forward) followed by his pronouncement, "You've got big troubles, my boy, the motor's in backwards." Ho, ho, ho.

Yes, Saabs are weird-and that's exactly why I like them. What other car company introduces a model like the venerable 900 and leaves it largely unchanged for more than a decade! Heck, everyone knows the thing to do is fiddle with the styling, change the name every couple of years, keep those showrooms hopping.

Whether it's the flight-deck interior (Saab makes well-respected commercial and military aircraft), the stubborn reliance on front-wheel drive and 4-cylinder turbocharged engines, or quirky mysteries like why the ignition key is between the front seats, Saabs have a unique personality. That's rare in these days of look-alike jelly beans when all cars seem to be designed by the same computer.

There's no denying I'm a Saabaholic. I got hooked in 1983 and corrupted my wife with a 1986 model. In fact, I'm a card carrying member of the local Saab club (Saabs Anonymous?). The club's monthly meetings are a good chance to shoot the breeze and swap stories with fellow travelers. Everyone claims their new setup, whether a hot box, sticky tires, or 96-octane fuel, is just the ticket.

This automotive equivalent of fish stories brings us to the silicon part of this story. My goal was to come up with an instrumentation setup which gives "no lie" comparisons of speed and handling tips and tricks. Photo 1

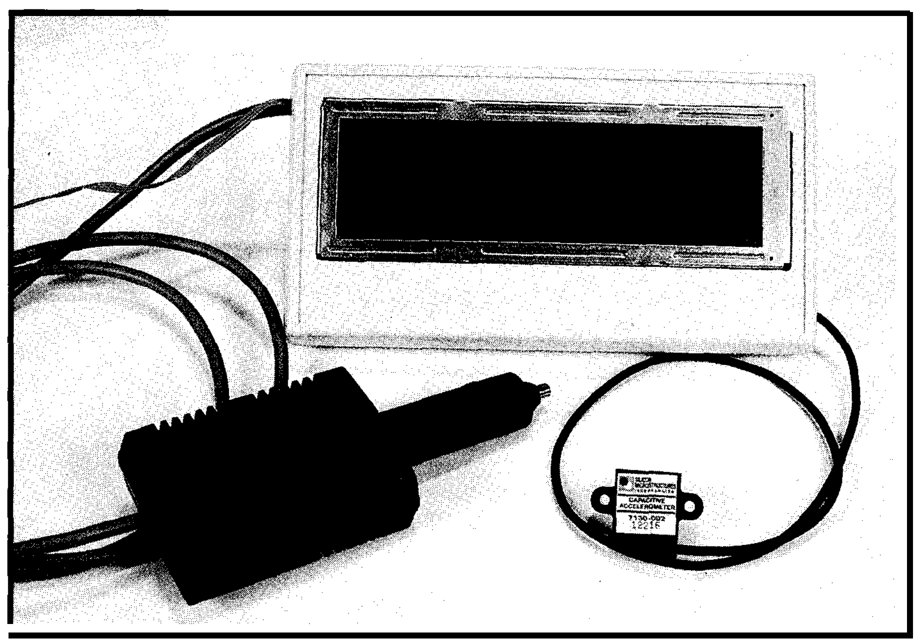


Photo 1-The "Speed Trap" system consists of a data logger (LCD + SBC), modified cigarette-lighter power supply and an accelerometer.

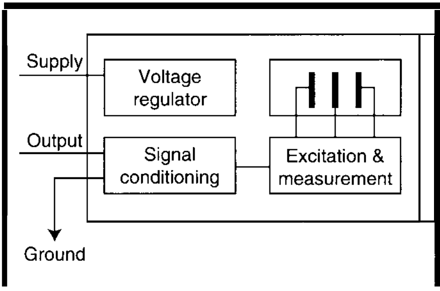


Figure l--The Silicon *Microstructures* 7130 *features* a simple *3-wire* interface: *power (9–20 V), ground, and a* 500-mV/g *output.*

shows the resulting and aptly named "Speed Trap" system. It consists of an LCD display and small SBC (in the box with the LCD) driven by one of those cigarette lighter DC power supplies (I modified the 4.5-V switch setting to produce 5 V by changing a resistor).

The key to the whole shebang is a gadget known as an *accelerometer* (the small black cube], specifically the 7130-002 from Silicon Microstructures. Before hitting the road, let's check under the 7130's hood.

NEWTON NABBER

An accelerometer measures that mysterious force called *gravity,* understanding of which came to Sir Isaac Newton (probably along with a headache) in an errant, apple-induced epiphany.

Philosophers still debate why gravity exists and where it came from even as many of Newton's concepts have been replaced by relativity on a cosmic scale. Nevertheless, at earthly velocities, the old F = MA (Force = Mass x Acceleration) still works fine for reality checking car enthusiasts' hypes and hopes.

Despite the common saying, gravity is an acceleration rather than a force. The unit of acceleration is known as g which, on Earth, happens to be about 32 feet per second^2. In other words, an object in free fall travels at 32 feet per second after one second, 64 feet per second after two seconds, and so on.

Accelerometers work on the same F = MA principle (i.e., a mass subject to an acceleration generates-thanks to inertia-a deflection force). By knowing the mass and measuring the force, acceleration can be determined.

Modern solid-state designs exploit silicon IC process techniques to micromachine tiny pendulums-truly amazing stuff! However, there are different techniques for measuring the deflection force that lead to a variety of subtle operational differences.

The simplest devices are piezoelectronic. Long-time readers may remember my article "Kynar to the Rescue" about piezo sensors (INK 22), which covers the wondrous properties of piezo material, best described as the molecular equivalent of a motor or generator. Like a motor, it can transform electrical input into physical work (e.g., piezo tweeters and our beloved quartz crystals). Of relevance in the current discussion, piezo materials also generate electrical output from work input, much as a motor can act as a generator.

Perhaps you've already guessed the inherent weakness of a simple piezoelectronic design-it can't handle DC (i.e., constant acceleration). For instance, you may remember the ACH04 from AMP (interestingly, they acquired the technology from the

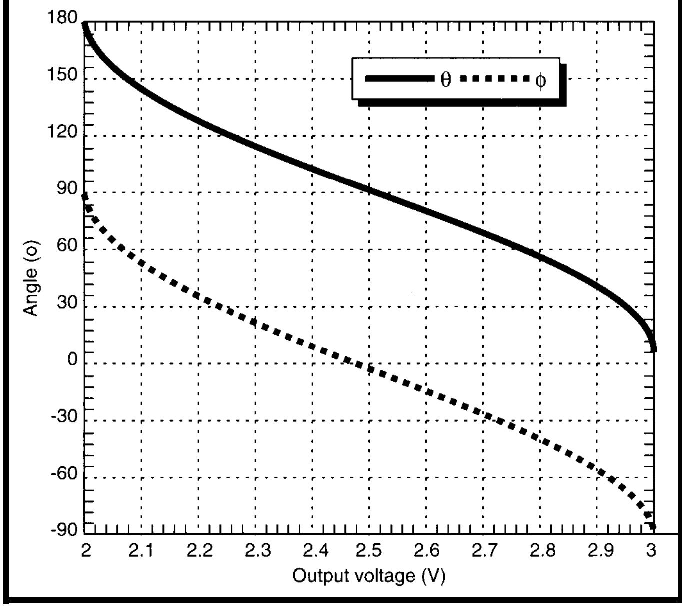


Figure 2b—*Knowing the effect* of *gravity on an inclined object* (Figure *2a),* the angle of *incline is easily determined as a function of* Vout *for both horizontal (upper curve) and vertical (lower curve) mounting.*

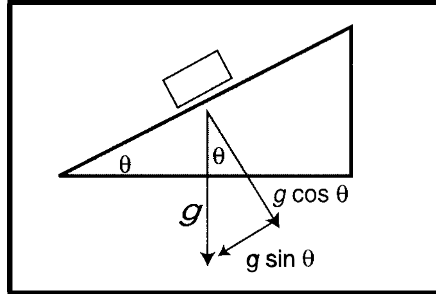


Figure Pa-An *accelerometer can* also *be used as an* inclinometer *if you apply some trigonometric relationships.*

Kynar folks) I mentioned in another article (*INK* 49). A close look at the data sheet shows that the output is only guaranteed down to 25 Hz.

To achieve DC frequency response, a variation on the theme is piezoresistive designs, which connect the mass to the package with the equivalent of strain gauges. However, there are a couple of problems to watch out for including temperature sensitivity (i.e., a thermistor) and the fact that external package-mounting forces tend to migrate inside and bias the response. Furthermore, though there are some exceptions, piezoresistive units usually offer low sensitivity (i.e., millivolts or microvolts per g), limiting them to high-g (100s, 1000s) shock detection. That's fine for applications such as airbag sensors, but definitely overkill for my test-drive plans.

Enter the latest technology-variable capacitance-of which the Silicon Microstructures unit is an example. These designs consist of a suspended mass and plate, with the gap between them changed by deflection of the mass-varying capacitance (see Photo 2).

The main claim to fame for variable-capacitance units like the 7130 is high sensitivity. As shown in Photo 3 and Figure l's block diagram, the unit combines a micromachined variable capacitor with support ICs (i.e., voltage regulator, calibration memory, signal condi-

tioner, etc.) to deliver a whopping 500 mV/g across a ±2 g range. The interface is blessedly simple, consisting of power (it's not fussy-anything within 9-20 V will do) and an output that varies from 1.5 to 3.5 V, with 0 g centered at 2.5 V.

The high-level output enables the logger to capture meaningful data, even with a lowly 8-bit O-5-V A/D converter. You might think more bits or some amplification (to expand the 2-V 7130 full-scale output to the A/D converter's S-V range) is called for, but in fact it works fine as is. The 8-bit A/D converter resolves down to about 0.04 g, which is a good match with the 7130's accuracy spec of 0.03 g.

This high-tech wizardry comes at a price-$225 for singles. However, in high volume [e.g., 10k), the chip [along with ±10, 50, 100, and 300 g cousins) approaches a more reasonable $50.

As an aside, note that an accelerometer that handles DC can work as an inclinometer in certain applications. As shown in Figure 2a, the acceleration vectors, acting on an inclined 7130, are easily derived with a little trig:

$$\text{Angle} = \text{Arccos}\left(\frac{\text{Vout} - \text{Vr}}{0.5} + 1\right)$$

where the angle and output of the arc cosine function are in degrees, and Vr is either 2.5 V or 3 V depending on whether the unit is mounted horizontally or vertically (i.e. 1 g or 0 g at rest-see Figure 2b).

The main restriction is that an accelerometer is only useful as an inclinometer when stationary-lest real acceleration get mixed in with the incline component. However, an accelerometer-based solution is ideal for harsh environments (i.e., shock or temperature extremes) in comparison to traditional floating-ball inclinometers.

AUTOMOTIVE BASICS

Listing 1 shows the main part of DRAG. B DT that runs the Speed Trap system. There is a second program, S H 0 K . B DT, but it's largely the same as the first part of DRAG (i.e., it captures the accelerometer data and graphs the g curve) and is thus not shown.

Listing I-The D RA G . B D T program records and displays acceleration, speed, and distance.

```
PROGRAM drag 'accleration. speed, distance'
INTEGER
      accel_data(2400),          'max 80 secs at 30 hz'
      idx,                       'index into accel_data'
      sample-time,               '#secs to sample-8,16,24,40,48,80'
      sample-count,              '#samples to take'
      scale,                     '#samples per pixel scale factor'
      speed_flag,dist_flag,      'flag 0-speed, distance times'
      x,y,x1,y1,                 'line start (x,y) and end (x1,y1)'
      i,j                        'int temp (for/next counters,etc.)'
REAL
      volts,                     'a/d reading'
      gs,prev_gs,                'volts -> g'
      speed,prev_speed,          'velocity in fps'
      zero_to_sx,                '0-to-speed time'
      dx_time,                   'time to distance'
      dx_speed,                  'speed at distance'
      mph,                       'fps -> mph'
      dist.                      'distance traveled'
      t                          'temp'
CONST
      cal0g=~130~,               '0g (virtual 2.5V) calibration'
      offset=-114-.              'centering factor'
      gain=-8-,                  'amplification factor'
      sample_rate=~30~,          '30 Hz'
      dx=-660-,                  '0-dx feet (ex:1320 ft=1/4 mile)'
      sx=-88-,                   '0-sx fps (ex:88fps=60mph)'
      max_mph=~100~,             'to scale vertical axis'

      screen_size=~240~,         '240 horiz. pixels'
      adc=~$9003~,               'a/d converter port addr'
      samples=-1-,               'name for sampling task'
      at_thirty_hz=~,2~,         'name for 30 Hz constant'
      BEEP=~?CHR$(7);~           'ring PC bell'

BEGIN 'drag'
  DO
   ?"Sample time (8,16,24,40,48,80 secs.)? ";:INPUT i
  UNTIL i=8 OR i=16 OR i=24 OR i=40 OR i=48 OR i=80

  sample_time=i
  scale=sample_time/8            'compute #samples/pixel'
  sample_count=sample_time*sample_rate 'compute #samples'

  GOSUB init                     'init a/d and lcd'
  idx=0                          'init pointer to log data'
  ?"Press a key to start logging...";
  i=KEY
  DO
     i=KEY
  UNTIL i<>0

  RUN samples at_thirty_hz       'dispatch sampler'
  BEEP
idle:                            'and wait until done'
  IF idx>=sample_count then GOTO ahead
  GOTO idle
ahead:                           all samples taken'
  BEEP
  CANCEL samples                 'so stop sampler task'
  x=7                            'start g curve at 8th dot'
  FOR i=0 TO screen-size-1       'for each pixel'
   y1=0
   FOR j=0 TO scale-1            'for each sample within pixel'
     y1=y1+accel_data((scale*i)+j)
   NEXT j
   y1=y1/scale                   'compute average accel'
```

(continued)

Listing 1-continued

```
    y1=y1-offset                  remove offset'
    y1=y1*gain                    amplify signal'
    y1=y1/4                       and scale to fit on lcd'
    y1=63-y1                      flip vertical so +g at top'
    IF i=0 THEN y=y1              start point for first line'
    x1=x+1                        move to next pixel'
    GOSUB line                    draw g curve on lcd'
    x=x1: y=y1                    set next line start'
  NEXT i

'Now compute distance and plot speed'
  speed_flag=0: dist_flag=0: speed=0: dist=0
  zero_to_sx=0: dx_time=0: dx_speed=0: prev_gs=0: prev_speed=0

  x=8: y=63                          'start speed curve at 0 mph'
  FOR i=0 TO screen-size-1           'for each pixel'
    FOR j=0 TO scale-1               'for each sample within pixel'
      volts=accel_data((i*scale)+j) * 0.0195 'a/d -> volts'
      volts = volts -(cal0g*0.0195)'adjust volts'
      gs = volts * 2                 'volts -> gs'
'compute velocity by integrating g curve (in fps)'
      speed = speed + (((prev_gs+gs)/2)*32.17405)/sample_rate
      prev_gs=gs
      IF speed-flag = 0 THEN BEGIN   'if not sx fps yet'
        IF speed >= sx THEN BEGIN    'then check for sx fps'
          zero_to_sx = (i*scale)+j     'if sx fps, log time'
          speed_flag=1             'and close log'
        END
      END
```

(continued)

The programs are written using BDT (BASIC Developers Tool), which is a high-level preprocessor for the SBC's built-in HD64180 BASIC-180. For a complete description of BDT, BASIC- 180, and the logger hardware and software (including the LCD drawing routines), refer back to "LCD Lineup-Getting Graphic With the LM213B" (INK 30).

Designing a data logger from scratch calls for a detailed signal-processing analysis, a fancy user interface with scrolling, zooming, scaling, and so on, and massive storage capability via hardware (e.g., flash card) and software (for data compression).

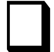
Then, there's the how you do it if you've only got a few days.. . .

Starting with the need for an accurate timebase quickly leads to the decision to rely on BASIC- 180's built-in multitasking. The tic rate is 60 Hz and the minimal multitasking program consists of a background program and a single task. Why, 30 Hz sounds grand to me-next question!

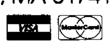
The only 8051/52 BASIC compiler that is 100 % BASIC 52 Compatible and has full floating point, integer, byte & bit variables.

- Memory mapped variables
- in-line assembly language option
- Compile time switch to select 8051/8031 or 8052/8032 CPUs
- Compatible with any RAM or ROM memory mapping
- Runs up to 50 times faster than the MCS BASIC-52 interpreter.
- includes Binary Technology's SXA51 cross-assembler & hex file manip. util.
- Extensive documentation
- Tutorial included
- Runs on IBM-PC/XT or compatibile
- Compatible with all 8051 variants
- **BXC51 $ 295.**

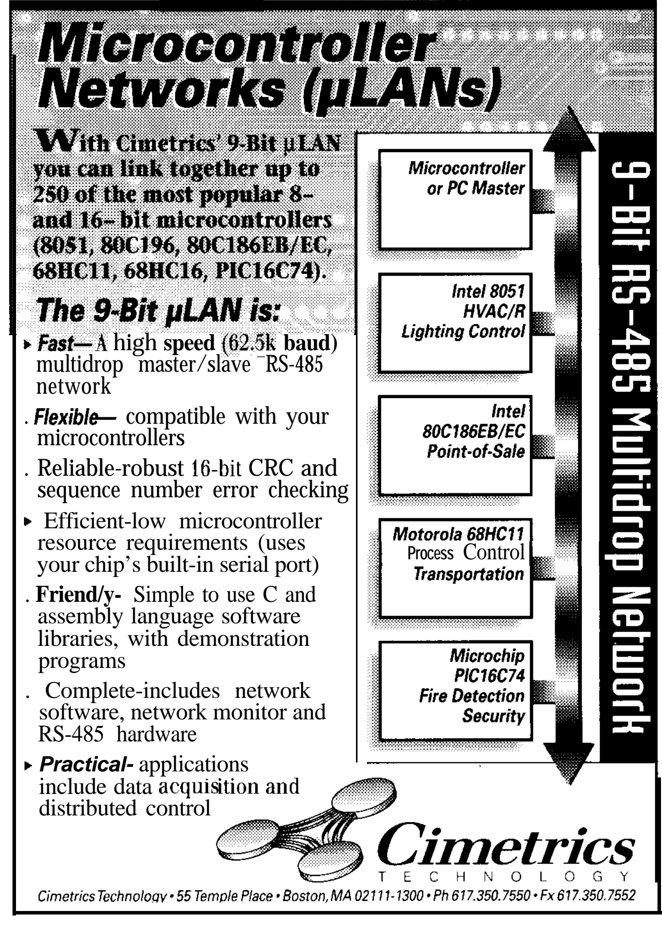
508-369-9556
FAX 508-369-9549



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



#129



#130

Designing a scrolling, zooming, and scaling GUI would be neat-some day. Instead, I simply cram everything on a single screen and empirically hardwire the scaling for a pleasing display. Deciding to use 240 of the LCD's 256 horizontal pixels, along with the 30-Hz spec suggests a minimum 8-s log time. With deadline looming, I'm quite open to suggestion.

The maximum logging interval required was scientifically determined to be 80 s since:

a] that's more than enough time to end up in the weeds and
b) 2400 elements is all that would fit in memory.

Of course, those who are committed (or should be) can pack the 8-bit A/D converter readings into a character string (rather than wasting the upper 8 bits of 16-bit integers) and use data compression (RLL is good and ADPCM, better) to boost storage.

The rather odd sequence of logging intervals (16, 24, 40, and 48 s) is the result of these decisions and the desire for a nicely spaced horizontal axis. Thus, all the logging intervals are integral divisors of 240.

I must remind you that as smart as the 7130 is, it is still analog and subject to analog's foibles. For instance, at first I fabricated a short adapter cable using phone wire. Firing everything up and running a few short tests showed a distressing amount of noise, perhaps ±50 mV—far worse than the 7130 accuracy spec.

A beginner would likely blame a bad sensor, A/D converter, or whatever. Being an old-timer, I quickly moved on to "what did I goof up this time?" Sure enough, connecting a known good power source quickly proved the noise wasn't coming from the 7130. Dispatching with the external wire in favor of an internal connection cleaned everything up.

Another set of concerns surrounds the issue of calibration. First of all, the logger and 7130 run on separate supplies (the former +5 V vs. unregulated +12 V for the 7130). It's not wise to assume that each unit has the same idea of what a volt is. Furthermore, the 7130 isn't totally impervious to temperature variations. There's a 2% drift in offset and span across 0-50°C.

Listing 1—*continued*

```
'compute dist by integrating velocity curve'
     dist = dist + ((prev_speed+speed)/2)/sample_rate
     prev_speed=speed
     IF dist_flag=0 THEN BEGIN 'if not dx feet yet'
       IF dist >= dx THEN BEGIN 'then check for dx feet'
         dx_time = (i*scale)+j  'if dx feet then log time'
         dx_speed = speed       'and speed'
         dist_flag=1            'and close log'
       END
     END
   NEXT j
   x1=x+1                       'set next line start'
   mph = (speed * 3600)/5280    'fps -> mph'
   t = (mph/max_mph)*63         'scale to fit on lcd'
   yl = 63-t                    'flip vert. so hi-speed at top'
   GOSUB line                   'draw speed curve on lcd'
   x=x1: y=yl
 NEXT I
 ?:?"0-";:?sx;:?"fps secs, ";:?dx;:?" ft. s,";:?dx;:?" ft. mph"
 ?zero_to_sx/30, dx_time/30, (dx_speed*3600)/5280

 ?"Press a key to dump accel_data..."
 DO
   i=KEY                        'wait for keypress'
 UNTIL i<>0
 ?sample_time
 ?sample_rate
 FOR i=0 TO sample_count-1
   ?accel_data(i)
 NEXT i
 DO
   i=KEY
 UNTIL i<>0
STOP

TASK samples
   accel_data(idx)=INP(adc)     'read a/d'
   OUT adc,0                    'start next conversion'
   idx=idx+1                    'next sample pointer'
EXIT
END 'drag'
```

So, I wrote a simple calibration program to repeatedly sample the ADC and average the results. Then, while running the program, I flipped the 7130 back and forth (i.e., label up, label down) expecting differences of 2 g (i.e., +1 g to -1 g). The results read from the A/D converter were (in decimal) +1 g = 156 and -1 g = 104. Dividing the difference by two yielded a virtual 2.5 V reading of 130 (versus the expected 127 or 128), which I plugged into the subsequent programs.

In principle, a calibration factor should be provided for the span, but my observed difference between + 1 g and -1 g was so close to ideal (i.e., 156 – 104 = 52 and 52 × 0.195 V = 1.014 V) that I didn't bother.

SHOCKING DISCOVERY

Over the years, I've added the bits and pieces (stabilizer bars, shocks, and springs) to my car that make up what Saab calls the SPG (Special Performance Group) handling package.

At fish story time, describing the handling differences between SPG and stock is limited to vague hand waving about "faster steering response," "less body roll," "not so floaty," and so on. The first tests were to document the SPG ride.

As mentioned, **SHOK. BDT** is essentially the same as the first part of **DRAG. B DT** so keep referring to Listing 1. Both programs start by enquiring for

the desired log interval (i.e., between 8 and 80 s) and then prompt for a keypress to start.

At that point, the `Samples` task is dispatched with the **Run** statement. If you look near the end of the listing, you'll see that the `Samples` task takes an A/D converter reading, stores it in the `accel_data` array, and increments the sample count (`idx`). Meanwhile, the background task sits in a loop, waiting for `idx` to reach the desired `sample_count`.

Once sampling is done, the `Samples` task is canceled and plotting of the results begin. For each pixel (remember, we've got 240 of them), I compute a result by averaging across the number of samples that compose that pixel. For instance, an 8-s log consists of 240 samples, so each reading is mapped directly. Longer intervals average a number of readings for each pixel (i.e., the total number of readings divided by 240). An 80-s log has 2400 readings, so 10 are averaged for each pixel.

Once the average is computed, a string of `y1=` statements mutates it into a y-axis pixel location between 0 and63. `offset` and `gain` are the empirically determined constants that make for a pleasing display (i.e., full scale and centered). Taking care to handle the special case of the first pixel (`IF i=0...`), a line is drawn between each pixel. Finally, the end of the current line makes the start of the next line in preparation for the next pass through the loop.

For a comparison, I pirated my wife's '86 with the stock suspension. Lest she worry needlessly, I adopted a minor subterfuge: "I think your fribblewumpus valve is making noise, dear. I'll check it for you."

The results show that the handling differences are real (Photos 4a–d). Seconds are notched along the horizontal access while the full vertical scale of the display (depending on the offset and gain constants) is about ±0.5 g.

Since these are only simple vertical g measurements, there isn't much to brag about. My wife summed it up in her own pithy way, "So you spent a bunch of money to make your car ride like a truck?" Testing the true benefits of the SPG package (100-MPH sweepers, skid pad, etc.) will only happen if headquarters agrees to cough up bucks for a set of tires and extra life insurance.

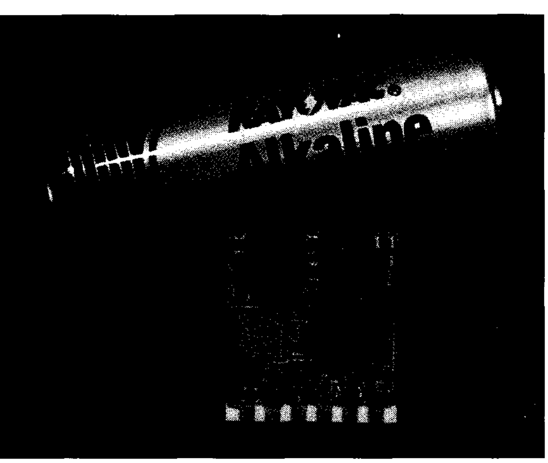


Photo 2—*The* 7130-002s *onboard support* circuits help *the* variable capacitance *g-sensor (silver* package) deliver *an* accurate *(±0.5%* typical), high sensitivity (500 *mV/g)* output.

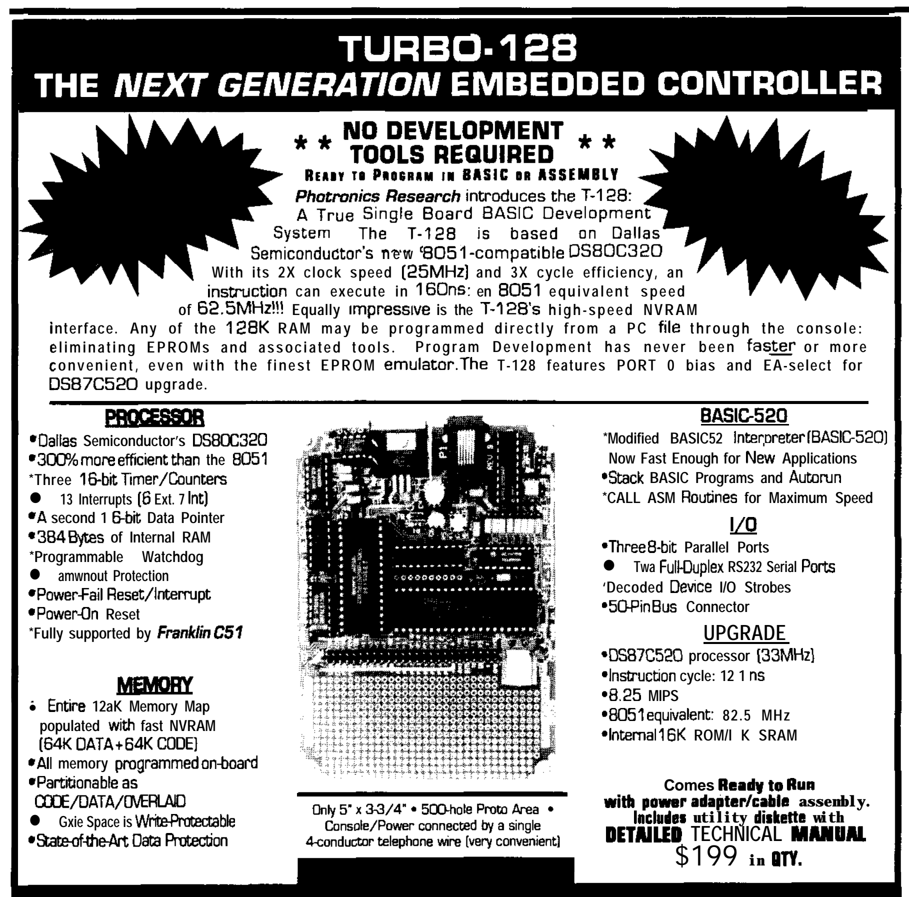
DAY AT THE RACES

DRAG.BDT continues on where **SHOK.BDT** leaves off. Remount the accelerometer with the label facing forward, lest you waste a run like I did. Unlike **SHOK**, **DRAG** computes actual gs, so take care to level the accelerometer relative to the road and not the car. I mounted the 7130 on a short-angle bracket that I could bend to account for a few degrees of rake.

Much as before, the remaining portion of **DRAG.BDT** steps through each pixel and each reading within the pixel. However, this time the result is converted to volts, adjusted with the 2.5-V calibration factor and turned into gs.

Knowing acceleration and time, it's simple to determine speed by multiplying the two and accumulating the result (in mathspeak, integrate using Simpson's rule). Knowing speed and time, a second integration yields distance. Along the way, the program tracks time to speed, time to distance, and speed at distance.

Next, the speed curve is plotted, with scaling determined by the `max_`



#131

mph constant (I used 100 so each of the 8 tics on the y-axis represents 12.5 MPH), and the run's results displayed.

Finally, the program waits for you to hit a key. It thendumps `accel_data` so you can capture it for plotting, printing, or further analysis. Just remember, "Any data logged can be used against you in court! "

Photos 5a and 5b compare the results of the S-speed turbo and nonturbo automatic. If you look closely, you can see the turbo run was marked by an exciting launch and a flattish g curve-the normal tendency for it to drop off is countered by the turbo kicking in. The program reported a 0-60-MPH time of 10.4 s. Subtract the half second it took to screw up my courage and it's remarkably close to the 10 s reported in the car mags.

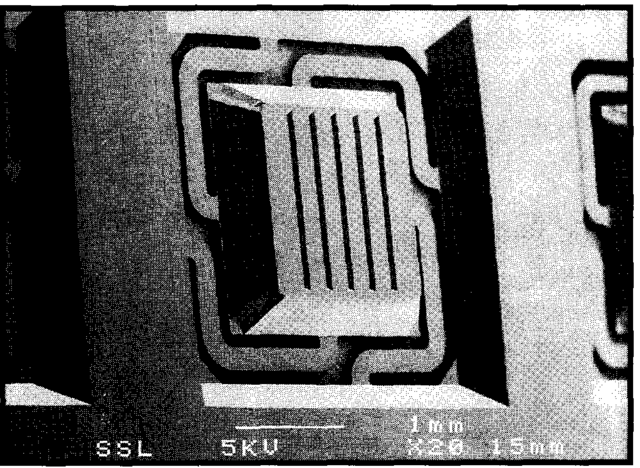


Photo 3—*The* 7130 uses *state-of-the-art micromachine techniques to fabricate the deflected mass.*

By contrast, the nonturbo run is, to put it politely, sedate with only the three-speed slushbox's valiant (but ultimately futile) lunge from first to second providing any excitement. Furthermore, it doesn't even live up to the official O-60 claims, indicating some maintenance is called for (when ***was*** the last time I checked that fribblewumpus valve?).

TICKET TO RIDE

Scene: Before dawn, a deserted thoroughfare in suburban Silicon Valley. ***Officer Speed has a Saab pulled*** over..

Officer Speed: You were exceeding the speed limit and having trouble staying in your lane.

Hapless Hacker: This may sound hard to believe, but I'm just gathering data on my car's performance. See, I write for this computer magazine called-heck, there's a copy in the glove box, so I'll just show..

Officer Speed: Keep your hands where I can see them. How much have you had to drink tonight?

Hapless Hacker: Oh heavens, I would never drive under the influence. I have enough trouble programming as it is. I can't even remember if `main` or the squiggly bracket is supposed to come fir.. .

Officer Speed: What's all that electronic equipment on the floor? You have receipts for that stuff?

Hapless Hacker: Well, er, uh, no..

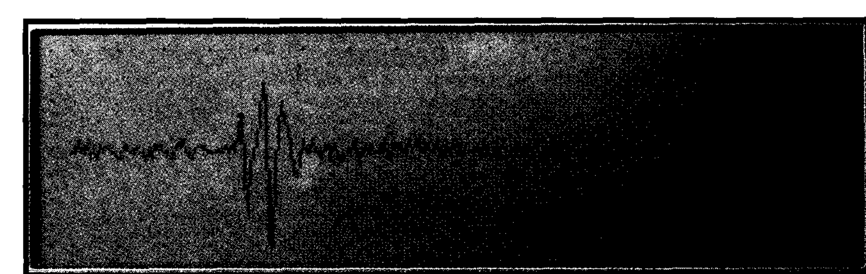
Officer Speed: Step out of the car..

Fortunately, it didn't happen to me, but it could happen to you. Worse, you might get hurt. Really worse, you might hurt someone else and that would truly be a "sob" story.

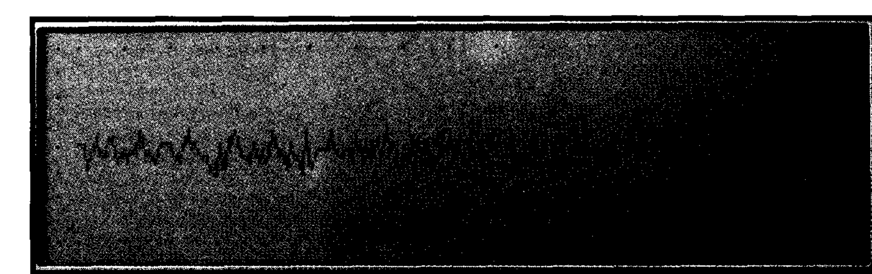
My four-foot stepladder has a total of eight (count 'em!) warning labels. Those cardboard sunscreens you stick in your windshield have fine-print warning "Do Not Drive With Sunscreen In Place." Somebody sued because their hot coffee was ***hot.*** [I

a)

b)



c)



d)

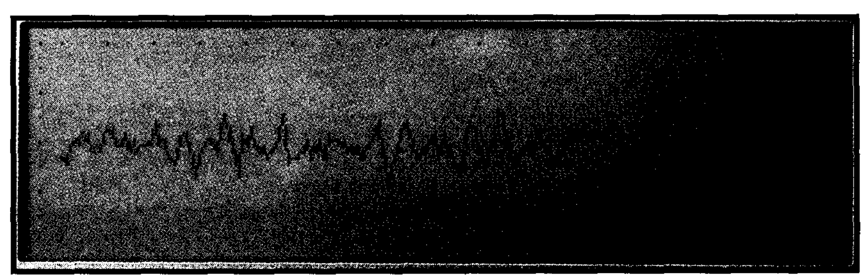
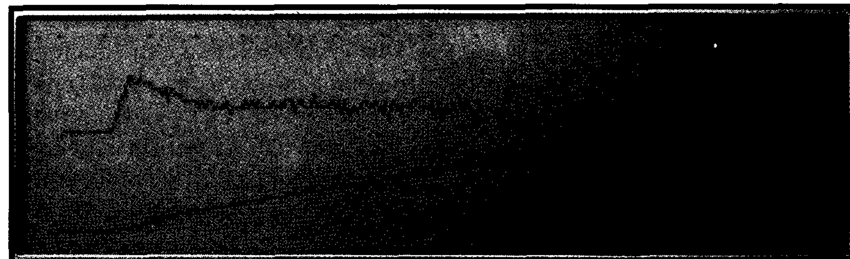


Photo *4-Driving over speeds bumps* with the *stock suspension (a) and* the *SPG (b) and driving along a rough road* with the stock *suspension (c) and* the *SPG (d) clearly document* that the latter *is much* stiffer.

a)



b)

Photo 5—*The* turbo has plenty *of power,* but calls for skilled launch technique to *exploit it* (a) *while the regular* 900 seems *to* need a *tuneup (b).*

wish I could sue for all the times I got "hot" coffee that wasn't.)

Thus, I feel obligated to issue the warning:

Don't debug and drive. Always remember to buckle up and backup.

That's it for now, gotta run. Tonight's the opening of the One-World Film Festival, and before that, I want to spend a few minutes downstairs with my new landing gear. ❑

Tom Cantrell has been an engineer 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.

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.

CONTA

Silicon Microstructures, Inc.
46725 Fremont Blvd.
Fremont, CA 94538
Attn: Jim Knell
(510) 490-5010
Fax: (510) 490-l 119

I R S

431 Very Useful
432 Moderately Useful
433 Not Useful

BCC52
BASIC-52
COMPUTER/CONTROLLER

The BCC52 controller continues to be Micromint's best selling single-board computer. Its cost-effective architecture needs only a power supply and terminal to become a complete development system or single-board solution in an end-use system. The BCC52 is programmable in BASIC-52, (a fast, full floating point interpreted BASIC), or assembly language.

The BCC52 contains five RAM/ROM sockets, an "intelligent" 2764/128 EPROM programmer, three 8-bit parallel ports, an auto-baud rate detect serial console port, a serial printer port, and much more.

PROCESSOR
- 80C52 8-bit CMOS processor w/BASIC-52
- Three 16-bit counter/timers
- Six interrupts
- Much more!

INPUT/OUTPUT
- Console RS232-autobaud detect
- Line printer RS-232
- Three 8-bit parallel ports
- EXPANDABLE!
- Compatible with 12 BCC expansion boards

MEMORY
- 48K RAM/ROM, expandable
- Five on-board memory sockets
- Either 8K or 16K EPROM

To Order Call 1-800-635-3355
Tel: (203) 871-6170
Fax: (203) 872-2204

BCC52	Controller board with BASIC-52 and 8K RAM	$1 89.00 Single Qty.
BCC52C	Low-power CMOS version of the BCC52	$199.00
BCC52I	-40°C to +85°C industrial temperature version	$294.00
BCC52CX	Low-power CMOS, expanded BCC52 w/32K RAM	$259.00

CALL FOR OEM PRICING

MICROMINT, INC. 4 Park Street, Vernon, CT 06066
in Europe: (44) 0285-658122 • in Canada: (514) 336-9426 • in Australia: (3) 467-7194
Distributor Inquiries Welcome!

#119

386 SBC $83 OEM (1K) PRICE
INCLUDES:
- 5 SER (8250 USART)
- 3 PAR (32 BITS MAX)
- 32K RAM, EXP 64M
- STANDARD PC BUS
- LCD, KBD PORT
- BATT. BACK. RTC
- IRQ0-15 (8259 X2)
- 0237 DMA 0253 TMR
- BUILT-IN LED DISP.
- UP TO 8 MEG ROM
- CMOS NVRAM

USE TURBO C, BASIC, MASM
RUNS DOS AND WINDOWS
EVAL KIT $295

$95 SINGLE PIECE PRICE
UNIVERSAL PROGRAMMER
- DOES 8 MEG EPROMS
- CMOS, EE, FLASH, NVRAM
- EASIER TO USE THAN MOST
- POWERFUL SCRIPT ABILITY
- MICROCONT. ADAPTERS
- PLCC, MINI-DIP ADAPTERS
- SUPER FAST ALGORITHMS

OTHER PRODUCTS:

8088 SINGLE BOARD COMPUTER .... OEM $27 ... $95
PC FLASH/ROM DISKS (128K-16M) 21 75
16 BIT 16 CHAN ADC-DAC CARD .... 55 ... 195
WATCHDOG (REBOOTS PC ON HANGUP) 27 95

- EVAL KITS INCLUDE MANUAL BRACKET AND SOFTWARE.
5 YR LIMITED WARRANTY
FREE SHIPPING
HRS: MON-FRI 10AM-6PM EST

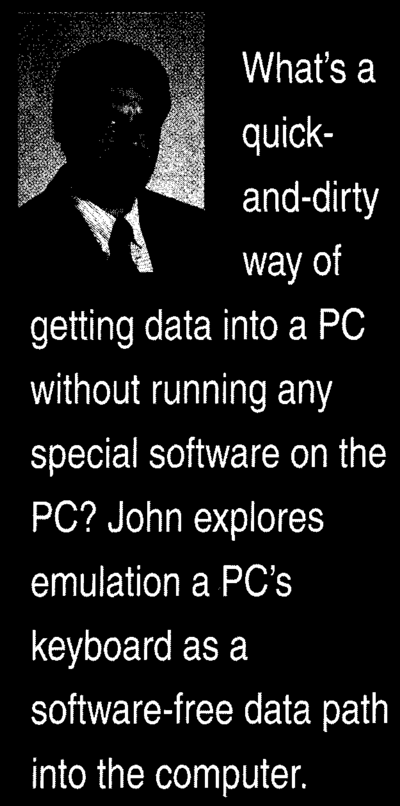
MVS BOX 850
MERRIMACK, NH
(508) 792 9507

132

EMBEDDED TECHNIQUES

John Dybowski

Using Keyboard I/O as an Embedded Interface



What's a quick-and-dirty way of getting data into a PC without running any special software on the PC? John explores emulation a PC's keyboard as a software-free data path into the computer.

Engineers are often charged with the task of making existing products do what they were never intended to do. This may be the natural consequence of a product's evolution or the result of taking the most expeditious path to developing something new.

When old designs become building blocks for another technology, it's effective to reuse as much of an existing design as possible. Under certain circumstance, there may be no other choice. This is certainly the case when using someone else's product as a component of a larger system.

In using other companies' products, the preferred course of action is, not unexpectedly, the one of least resistance. Faced with the task of hooking a new peripheral to an existing system, you might consider structuring the new device's support code to look like the one it's replacing.

The attraction of this common programming trick is that it disturbs the functioning code as little as possible. This sort of deceptive programming is really what device drivers are all about.

DECEPTIVE DESIGN

Consider a thoughtfully designed display driver as an example. With a defined method of passing input and output arguments and generic functions for standard operations such as device initialization, the main program could care less if the actual output device was an LCD, vacuum fluorescent panel, CRT terminal, or anything else with a similar set of features.

At the other extreme, the consequences of embedding device dependencies directly into your process code can prove to be intolerably restrictive should a change be ultimately required. The fact is, even if you are content to specify a specific peripheral for a given application, who's to say the manufacturer will be able to deliver a year from now. Stuff happens.

This is not to say that incorporating device-specific functions in your code is necessarily bad. Certainly, there are cases where efficiency dictates that we stray from the ideal—a system processor doesn't have the required throughput for a heavily layered device-support structure. However, be aware of the tradeoffs. Simply put, it's okay to write bad code if there's no other way to do it.

This may seem to be quite a foolish statement. But realize that there are definitely a number of very popular processors that make writing good code very difficult, if not impossible. This is especially true of some of the very low-end controllers that present a varied and unending assortment of ways to cramp your style-a meager and irregular instruction set, bizarre program-memory paging schemes, or diabolically restricted addressing modes. Sound familiar?

Device drivers essentially offer a stylized means of defining parameter-passing conventions between program modules. The inherent benefits, however, shouldn't be limited to modules with a formally defined interface specification.

It's often equally advantageous to use this programming technique within the depths of your program for changing the way calls and inline functions operate. When faced with the prospect of replacing major function blocks, I go out of my way to make the new code imitate the original's I/O conventions. The goal: don't let the main program know there's a difference.

This type of imitation, applied to hardware devices, is commonly referred to as *emulation.*

EMULATE THIS

A standard device that can be emulated advantageously is the IBM keyboard. By knowing how the IBM keyboard electronics and communication protocol are structured, a wide variety of equipment can be compelled to function as either a sending or a receiving device.

The wide availability of very small IBM-compatible computers designed for embedded applications extends the potential for this standard interface. It would be an advantage to use the BIOS keyboard-support services regardless of the form your keypad took. Alternatively, although unquestionably of less utility, you might connect your proprietary embedded controller to a standard PC keyboard.

Another possible use for keyboard emulation may not have anything to do with a keyboard at all. Consider an application where you have to enter data collected by an embedded instrument into a spreadsheet or some other program running on your desktop PC. Since most data collection devices have some sort of serial interface, you might be tempted to first obtain a printout and enter it into your computer manually. Although requiring the least upfront work, this approach has the disadvantage of being time consuming and error prone.

A step toward automating the procedure may involve capturing the serial data to a file for later processing. Or, you might feel creative and write a little TSR that intercepts data from the serial port and deposits it into the PC's keyboard buffer. This back-door approach could save you the intermediate steps otherwise required in preparing your filed data for input into your PC program.

A more direct approach simply makes the data look like it's coming from a keyboard in the first place, bringing it through the keyboard port.

Obviously, this method can only be applied to a limited number of data-collection tasks. If it is suitable for the volume and the nature of the data you're manipulating, there are several distinct advantages. Not to be underestimated is the fact that all your alterations are made far away from the stuff that's already running!

STANDARDKEYBOARDS

The IBM-compatible keyboard has gone through two transformations: the PC/XT and the AT keyboards. As you'd expect, both designs use built-in microcontrollers to manage the matrix scanning and handle communications to the computer. The original PC/XT computer used nothing more than a shift register and flip-flops to receive data transmitted by the keyboard.

Starting with the AT, the computer interface consists of a slave microcontroller that acts as an intermediary for all bidirectional communications to and from the keyboard. The added capability of the AT's keyboard-communication processor opens up the potential for managing a lot more complexity in the keyboard-communication protocol.

Although the PC/XT-style keyboard has deficiencies, it also has (of necessity) the virtue of simplicity. With this in mind, let's see what the two types have in common before examining the older design and how it was transformed into the ubiquitous AT configuration.

The concept common to both PC/XT and AT keyboards is that physical key scanning is carried out under control of an onboard microcontroller. The controller detects when a key is pressed and released and sends this information to the computer.

Rather than outputting standard ASCII codes, IBM keyboards attain greater flexibility by transmitting make and break scan codes as keys are pressed and released. These scan codes are assigned by numbering the physical keys on the original PC/XT keyboard from left to right, top to bottom. It's up to the computer's BIOS to convert these unique codes into ASCII codes when possible. Special keys that don't have corresponding ASCII symbols are given a null value followed by the scan code. This null causes the computer to properly interpret the following code as a scan code rather than an ASCII code.

Communication between the keyboard and computer is accomplished over a data line and a clock line. These lines are driven by open-collector devices with their associated pull-up resistors at either end. A typical keyboard line interface is depicted schematically in Figure 1.

THE PC/XT KEYBOARD

Figure 2 shows the original PC/XT computer's keyboard interface. As you can see, this hardware implementation centers around a shift register and several flip-flops. Data bits

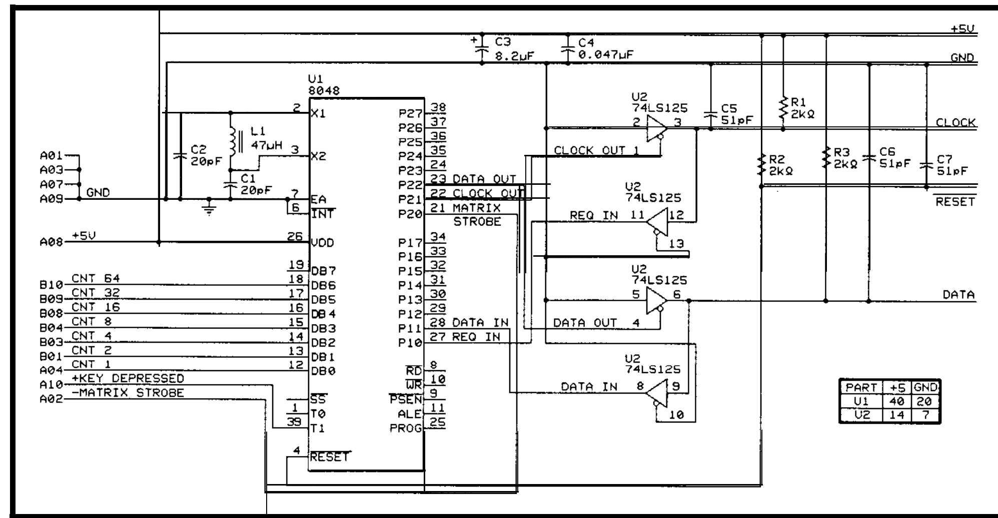


Figure I--The circuitry *inside the PC/XT's keyboard (shown here) is very similar to that in an AT keyboard. The main differences are in the firmware.*

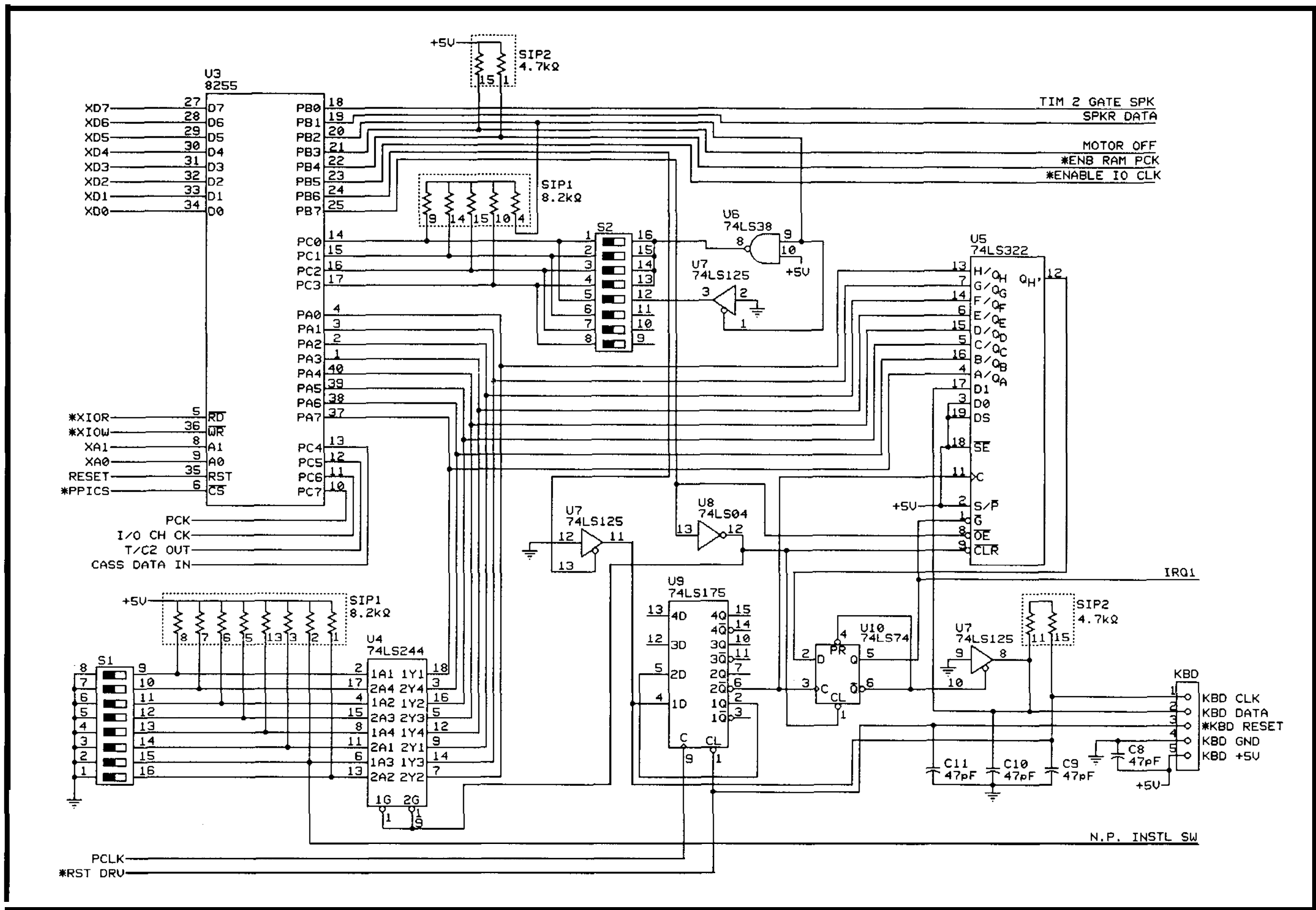


Figure 2—The original IBM PC's keyboard port consisted of just a shift register and some glue logic and could only receive data from the keyboard.

are shifted in on falling edges of the keyboard-generated clock and are valid from before this falling clock edge until after the rising edge of the clock.

The data line registers a high start bit and eight data bits for each transmitted code. Looking again to Figure 2, you see that initially a reset signal clears the shift register and its associated flip-flop. As data bits are clocked in, the high start bit propagates through the shift register and appears at the ninth-bit flip-flop. This asserts an interrupt on the CPU.

At the same time this flip-flop pulls the clock line low. This signals a busy state to the keyboard, which is held until the received character has been processed by the computer. Once this busy status clears, the clock line is released and is pulled high by the pull-up resistor. This indicates to the keyboard that the interface is available for further transmissions.

Curiously, with the interface in its idle state, the keyboard lets the clock line pull high, but drives the data line low. This turns out to be an unfortunate decision on the part of the design engineers for it essentially jams the data line, making bidirectional communication impossible.

As a result, it's not possible to have multiple transmitting devices on the line without adding extra circuitry to minimally disconnect the keyboard's data line from the rest of the interface. Because of the way make and break codes are represented, the PC/XT keyboard is capable of encoding 128 different key codes. A byte with a value of O-127 is a make code. Adding 80 hex to this basic code creates a break code. For example, if Olh is the make code, 81h is the break code.

THE AT KEYBOARD

The AT keyboard rectifies some shortcomings of the PC/XT keyboard while introducing a level of complexity that seems a bit out of place in a keyboard. Remember that the AT computer's keyboard port uses a dedicated microcontroller to manage all communications with the keyboard. This explains why things get complicated. Simply put, with the aid of this additional processing power, getting complicated is easy to do.

The AT interface is defined as bidirectional. The data line is now left pulled up while no data is being transmitted or received so either the keyboard or the computer can take control of the interface during idle times. As I'll show later, this also leaves the possibility of adding other external devices that can easily seize control of the interface.

Bidirectional data communication between the keyboard and computer consists of 1 l-bit datastreams composed of a low start bit, eight data bits, an odd parity bit, and a high stop bit. The clock and data relationship remains similar to that of the PC/XT. A fairly comprehensive [for a keyboard) protocol is defined that provides

for error detection and retransmission, abort timing, and a line-contention recovery. Additionally, a relatively complete command set is specified that describes a number of useful (and not so useful) functions that the keyboard and computer can initiate.

As with the PC/XT keyboard, all keys are handled on a make/break basis. The difference is that to handle more than 128 keys, a different method of denoting break codes is used. Break codes are transmitted as F0h followed by the hex make code. Now a make and break sequence for scan code 1 appears as 01h, F0h, 01h.

The AT's keyboard-interface electronics are very similar to those used in the PC/XT. The computer's interface is implemented in firmware running on a microcontroller, so it's pointless to try to depict it schematically. It's just your typical black box.

NEGOTIATING THE WIRE

With both the PC/XT- and AT-style keyboards, all communications are carried out using open-collector drivers on the clock and data lines. With the PC/XT, there's not much more to a typical transaction than what I've already said. This is partially due to the fact that the interface is capable of unidirectional traffic only and in part because you can only get into so much trouble with a shift register and some glue.

The situation is a little more interesting with the AT. I'll elaborate more fully on how the keyboard and computer negotiate for control of the line and what happens if there's a conflict. As stated, when no communication is occurring, the data and clock lines are held at a high level through pull-up resistors. The use of open-collector drivers allows either end to assert a logic low on either of the interface lines.

If the computer is doing something, it may elect to hold off a keyboard transmission by pulling the clock line to a low level (inhibit status). In a similar fashion, the computer signals its intention to begin transmitting by asserting a low level on the data line while leaving the clock line at a high level (RTS status).

If either condition is in effect, the keyboard will not attempt to transmit. Note that when the computer asserts request-to-send (data line low), it puts its start bit on the line. On recognition of this event, the keyboard proceeds by emitting 11 clocks. The first 10 strobe in the start bit, eight data bits, and the parity bit. After the tenth bit, the keyboard pulls the data line low and issues one final clock pulse. This keyboard-generated stop bit signals the computer that the keyboard has received the transmission. The computer returns to a ready state or puts the interface into inhibit status.

The keyboard checks for inhibit status and request-to-send status prior to starting any data transmission. Once a transmission has been initiated, the keyboard must continuously check the clock. Should the computer lower the clock line while the keyboard is transmitting, the interface enters a state called *line contention.*

What happens now depends on how far into the sequence the keyboard is. If this contending state is recognized before the rising edge of the tenth clock (the parity bit), the keyboard releases the clock and data lines and retains the pending data for later retransmission. If line contention occurs after the tenth bit, the transmission is assumed to have "gone through" and the transfer concludes normally. On receipt of the keyboard's data, the computer puts the interface into inhibit status if it needs extra processing time or a response request is to be issued.

US VERSUS THEM

When IBM developed the AT computer, it's obvious they had the resources to put together a design team just to handle the keyboard design. For those of us with lesser means, it's important to separate the essential from the superfluous. That is, we have to cut through the fluff. Through empirical observation, some generalizations about the operation of the AT keyboard can be made.

For example, for most keys, only their respective make codes are significant. The complementary break codes are unessential and make no

If you need I/O...

Add these numbers up:

80C552 a '51 Compatible Micro
40 Bits of Digital I/O
8 Channels of 10 Bit A/D
3 Serial Ports (RS-232 or 422/485)
2 Pulse Width Modulation Outputs
6 Capture/Compare Inputs
1 Real Time Clock
64K bytes Static RAM
1 + UVPROM Socket
512 bytes of Serial EEPROM
1 Watchdog
1 Power Fail Interrupt
1 On-Board Power Regulation

It adds up to real I/O power!

That's our popular **552SBC**, priced at just $299 in single quantities. Not enough I/O? There is an expansion bus, too! Too much I/O? We'll create a version just for your needs, and pass the savings on to you! Development is easy, using our Development Kit: The 552SBC-50 Development board with ROM Monitor, and an 8051 C compiler for just $449.

New Versions of the 8031SBC

Our popular 8031SBC can now be shipped with your favorite 8051 family processor. Models include **80C51 FA, DS80C320, 80C550, 80C652, 80C154, 80C851** and more. Call for pricing today!

Truly Low-cost In-Circuit Emulator

The DryICE Plus is a low-cost alternative to conventional ICE products. Load, single step, interrogate, disasm, execute to breakpoint. Only $448 with a pod. For the 8051 family, including Philips and Siemens derivatives. Call for brochure!

Call for your custom product needs. Quick Response.

HiTech Equipment Corp.
9400 **Activity** Road
San Diego, CA 92126
(Fax: (619) 530-1 **458**)

Since 1983

(619) 566-1892

Internet e-mail: info@hte.com
Internet ftp: ftp.hte.com

#133

difference to the computer. It appears the PC BIOS only uses the break codes for keys such as Alt, Shift, and Ctrl. This may not be particularly significant if all you're doing is developing a compatible keyboard. If, on the other hand, you are translating ASCII data to emulate a data stream that merely has to look like it's coming from a keyboard then this knowledge can save a lot of unnecessary line traffic, not to mention wasted code.

And speaking of traffic, a lot of the protocol's intricacy exists to handle data errors, count abort times, and sort out line contentions. Somewhat associated to this is the command set that defines a multitude of functions the keyboard and computer are to support. Adherence to the rigors of the specification depends on what you're trying to accomplish. Experience shows that just clocking your data across the interface yields satisfactory results for a wide range of applications.

Regardless of the degree to which you're intending to emulate a real keyboard, there are several issues you should carefully consider. For instance, there are a number of pitfalls if your emulation device must operate in conjunction with a "live" keyboard. Keeping track of the state of the keyboard/computer interface is not something you want to take lightly. Recall that certain key codes are capable of causing the system BIOS to effectively redefine the attributes of the majority of the keys.

For example, consider that your emulation device inadvertently seizes the interface after a Ctrl make code is sent by the keyboard. Obviously, the data ultimately seen by the PC application differs substantially from what you intended.

It would perhaps be even more disturbing to corrupt the Ctrl break because of a data collision. Here, hopefully the communication protocol would bail you out.

You might consider providing special lockout circuitry on your interface that disconnects the keyboard from the computer when your device is sending data. Here again, you could potentially get in trouble if certain make/break sequences were

Listing 1—*A rudimentary AT keyboard driver performs fine for programs that use BIOS keyboard I/O.*

```
;PUBLIC ENTRY POINT
        PUBLIC  AT_XMIT

;EXTERNAL REFERENCES

        EXTRN   BIT (KEY_CLK)
        EXTRN   BIT (KEY_DATA)
        EXTRN   BIT (SEND_EXT)

;CONSTANTS

SHIFT   EQU     12H
CONTROL EOU     14H

BREAK   EQU     0F0H

;ASSEMBLE INTO CODE SEGMENT
;
PROG    SEGMENT CODE
        RSEG    PROG

;ROUTINE TO SEND DATA STRING TO AT VIA KEY PORT
;INPUT: RO=BYTE COUNT
        R1=POINTER FOR IRAM SOURCE (IF SEND_EXT=0)
        DPTR=POINTER FOR XRAM SOURCE (IF SEND_EXT=1)
;
AT_XMIT:
        MOV     A,R0
        JNZ     ATX0            ;NOTHING TO SEND?
        RET

;MAIN TRANSMIT LOOP

ATX0:
        JB      SEND_EXT,ATXI   ;EXTERNAL SOURCE?
        MOV     A,@R1           ;GET ASCII
        INC     R1
        SJMP    ATX2
ATX1:
        MOVX    A,@DPTR         ;GET ASCII
        INC     DPTR
ATX2:
        MOV     B,A             ;SAVE ASCII
        CALL    XCHAR           ;TRANSLATE
        JZ      ATX5            ;CONTROL?
        CJNE    A,#0FFH,ATX3    ;SHIFTED?
        SJMP    ATX4

;NOT SHIFTED CHARACTER

ATX3:
        CALL    XMIT            ;SEND
        DJNZ    RO,ATXO         ;DONE?
        RET

;SHIFTED CHARACTER

ATX4:
        MOV     A,#SHIFT        ;SHIFT ON
        CALL    XMIT
        MOV     A,B             ;RETRIEVE ASCII
        CALL    XSHFT           ;TRANSLATE
        CALL    XMIT            ;SEND CODE
        MOV     A,#BREAK        ;RELEASE
        CALL    XMIT
        MOV     A,#SHIFT        ;SHIFT OFF
        CALL    XMIT
```

(continued)

Listing 1-continued

```
        DJNZ    R0,ATX0         ;DONE?
        RET

;CONTROL CHARCTER
;
ATX5:
        MOV     A,#CONTROL      ;CONTROL ON
        CALL    XMIT
        MOV     A,B             ;RETRIEVE ASCII
        CALL    XSHFT           ;TRANSLATE
        CALL    XMIT            ;SEND CODE
        MOV     A,#BREAK        ;RELEASE
        CALL    XMIT
        MOV     A,#CONTROL      ;CONTROL OFF
        CALL    XMIT
        DJNZ    R0,ATX0         ;DONE
        RET
;
;ROUTINE TO SEND A CHARACTER TO AT KEY PORT
;LOCAL REGISTER USAGE: R2/R4=DELAY LOOP COUNTER
;                      R3=BIT COUNTER
XMIT:

;INTERCHARACTER DELAY FIRST

        CALL    DELAY

;START BIT
```

(continued)

disassociated. In such a situation, an undetected, lost keyboard transmission would be even more likely.

With a little extra hardware you could hold off the keyboard while you are transmitting by asserting an inhibit status on the [now isolated) keyboard interface. In such a scenario, constant line monitoring of all transmissions from the keyboard and computer would be necessary.

What all this boils down to is that you require a little smarts of the person operating the equipment or you can put a lot of smarts in your equipment. Sometimes, the former constitutes an unreasonable assumption. Then again, you have to give realistic consideration to the system's operating conditions. The fact might be that in some cases a person would really have to work to break it.

Putting things into perspective, the situation I described is not nearly as much of a problem in a typical system implementation as would seem at first. However, looking at things on a detailed level flags the hazards of a particular approach. Walking the line between too much and too little is always difficult and often involves subjective judgment calls. Sure, you can render a design to handle all eventualities, but you might price your product right out of the market.

KEY CODE AND UNEXPECTED APPLICATIONS

In keeping with my usual premise of starting simple, I'll present a rudimentary transmit-only AT-keyboard emulation driver. For flexibility, the code includes an ASCII–to-scan-code translator.

As an interesting side note, this code is similar to something I developed several years ago as part of a remote computer-control center. It turns out that some of the most intriguing applications may be totally unexpected and unforeseen, which is exactly what happened. Through a fortunate sequence of events, one of my systems eventually found its way into the hands of some rather inventive software developers. It was just the thing they were looking for.

#1

FREE
Data Acquisition Catalog

1995

PC and VME data acquisition catalog from the inventors of plug-in data acquisition. Featuring new low-cost A/D boards optimized for Windows, DSP Data Acquisition, and the latest Windows software. Plus, informative technical tips and application notes.

Call for your free copy

1-800-648-6589

ADAC

American Data Acquisition Corporation
70 Tower Office Park, Woburn, MA 01801
phone 617-935-3200 fax 617-938-6553

#135

Listing 1-continued

```
        CLR     KEY-DATA        ;DATA LOW
        NOP
        CLR     KEY_CLK         ;CLOCK LOW
        MOV     R2,#5
        DJNZ    R2,$
        SETB    KEY_CLK         ;CLOCK HIGH
;
;SETUP FOR LOOP

        MOV     R3,#9           ;BIT COUNTER
        MOV     C,P             ;ODD PARITY
        CPL     C

;MAIN BIT BANGING LOOP
;
CX1:
        RRC     A
        MOV     KEY_DATA,C      ;SETUP DATA
        NOP
        CLR     KEY_CLK         ;CLOCK LOW
        MOV     R2,#5
        DJNZ    R2,$
        SETB    KEY_CLK         ;CLOCK HIGH
        DJNZ    R3,CX1          ;DONE?

;SEND STOP BIT

        SETB    KEY-DATA        ;DATA HIGH
        NOP
        CLR     KEY_CLK         ;CLOCK LOW
        MOV     R2,#5
        DJNZ    R2,$
        SETB    KEY_CLK         ;CLOCK HIGH
        RET

;10-ms INTER-CHARACTER DELAY ROUTINE
;
DELAY:
        MOV     R4,#20
DELAY1:
        MOV     R2,#0F8H
        DJNZ    R2,$
        DJNZ    R4,DELAY1
        RET

;PRIMARY TRANSLATE ROUTINE
,
XCHAR:
        INC     A
        MOVC    A,@A+PC
        RET

;PRIMARY LOOKUP TABLE
;0=CONTROL CHARACTER, FF=SHIFTED

        DB      000H,000H,000H,000H,000H,000H,000H,000H
        DB      066H,00DH,000H,000H,000H,05AH,000H,000H
        DB      000H,000H,000H,000H,000H,000H,000H,000H
        DB      000H,000H,000H,000H,000H,000H,000H,000H
        DB      029H,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,052H
        DB      0FFH,0FFH,0FFH,0FFH,041H,04EH,049H,04AH
        DB      045H,016H,01EH,026H,025H,02EH,036H,03DH
        DB      03EH,046H,0FFH,04CH,0FFH,055H,0FFH,0FFH
        DB      0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH
        DB      0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH
        DB      0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH
        DB      0FFH,0FFH,0FFH,054H,05DH,05BH,0FFH,0FFH
```

(continued)

These people had written a rather complex PC application and were adamant about testing it thoroughly before permitting even a beta release. Manually testing from the keyboard was not deemed feasible due to the number of input permutations and the likelihood of operator error.

At the same time, they were skeptical about the possibly disruptive effect of performing the final testing using the special TSR they had used during the initial check out. In short order, they had written a special program designed to run on a dedicated computer that operated the remote computer-control system as a robot typist. The test sequence ran at maximum speed for 24 hours a day over a period of three weeks! Due in great part to this test plan, when the software shipped, it was remarkably bug free.

Anyway, back to the code. Listing 1 is the AT-keyboard emulation driver presented in its entirety. The program includes both an ASCII-to-scan-code translation algorithm and a bit-banged scan-code transmission routine. This 8031 assembly-language code can accept its input string in either internal or external RAM. The character count is passed in register R0, the internal RAM data pointer uses R1, and the external RAM data pointer uses DPTR. The bit variable SEND_EXT determines whether the input string resides in internal or external RAM.

On entry, R0 is checked. If it does not contain 0, (an errant null string), the code falls through to the main data-transmission loop. Now a data byte is picked out of the appropriate data area in accordance with the SEND_EXT bit flag.

The initial ASCII translation is performed by XCHAR, which returns either the actual scan code or a special indicator. If the returned value is neither 00h nor FFh, then no further processing is required. The scan code is dispatched to XMIT and is clocked out to the computer. A returned value of FFh indicates the ASCII character requires a shift operation whereas a 00h means the ASCII character involves a control operation.

Listing 1—*continued*

```
        DB      00EH,01CH,032H,021H,023H,024H,02BH,034H
        DB      033H,043H,03BH,042H,04BH,03AH,031H,044H
        DB      04DH,015H,02DH,01BH,02CH,03CH,02AH,01DH
        DB      022H,035H,01AH,0FFH,0FFH,0FFH,0FFH,071H

;SECONDARY CHARCTER TRANSLATE ROUTINE
;
XSHFT:
        INC     A
        MOVC    A,@A+PC
        RET
;
;SECONDARY LOOKUP TABLE

        DB      01EH,01CH,032H,021H,023H,024H,02BH,034H
        DB      033H,043H,03BH,042H,04BH,03AH,031H,044H
        DB      04DH,015H,02DH,01BH,02CH,03CH,02AH,01DH
        DB      022H,035H,01AH,054H,05DH,05BH,036H,04EH
        DB      029H,016H,052H,026H,025H,02EH,03DH,052H
        DB      046H,045H,03EH,055H,041H,04EH,049H,04AH
        DB      045H,016H,01EH,026H,025H,02EH,036H,03DH
        DB      03EH,046H,04CH,04CH,041H,055H,049H,04AH
        DB      01EH,01CH,032H,021H,023H,024H,02BH,034H
        DB      033H,043H,03BH,042H,04BH,03AH,031H,044H
        DB      04DH,015H,02DH,01BH,02CH,03CH,02AH,01DH
        DB      022H,035H,01AH,054H,05DH,05BH,036H,04EH
        DB      00EH,01CH,032H,021H,023H,024H,02BH,034H
        DB      033H,043H,03BH,042H,04BH,03AH,031H,044H
        DB      04DH,015H,02DH,01BH,02CH,03CH,02AH,01DH
        DB      022H,035H,01AH,054H,05DH,05BH,00EH,000H

        END
```

The shift code is handled by first outputting a shift make code. A secondary lookup using XSHFT translates the original ASCII code to a scan code. This code is transmitted, followed by a break code and a shift code (shift break sequence). The procedure for a control code is similar, except the sequence is framed with a control scan code. These basic steps repeat until the character count in R0 is exhausted.

The code works okay, but is not without its problems. No attempt is made to determine the actual operational status of the keyboard-to-computer interface. The rather flagrant assumption is made that the interface is in normal mode-that is, not in a state such as Shift, Ctrl, Alt, Caps Lock, and so on.

Also, note that strings involving a lot of shifted sequences suffer a significant performance penalty since each ASCII character results in the transmission of three codes in addition to the translated scan code: a shift, break, and shift. Obviously, if I were to improve this program, keeping track of my own shift status would be one of the first areas I'd address.

Next month, I'll wrap up with a discussion of the AT-keyboard command set, scan-code tables, and a few other details I was forced to omit this month because of space constraints. I'll conclude with a demonstration of a real application based on the material presented.

John Dybowski is an engineer involved in the design and manufacture of embedded controllers and communications equipment with a special focus on portable and battery-operated instruments. He is also owner of Mid-Tech Computing Devices. John may be reached at (203) 684-2442 or at john.dybowski@circellar.com.

IRS

434 Very Useful
435 Moderately Useful
436 Not Useful

CONNECTIME

conducted by Ken Davidson

The Circuit Cellar BBS
300/1200/2400/9600/14.4k bps
24 hours/7 days a week
(203) 871-1988—Four incoming lines
Internet E-mail: sysop@circellar.com

It's been a busy month of upgrades on the BBS. We received a new version of the BBS software *that completely* replaces the file section interface *with one that allows full-screen selection of* files *for batch* downloads. There are lots of other small improvements as well.

Our *Internet provider also upgraded their pipeline from a 56k* line *to a T1. We should see an improvement in mail and newsgroup delivery times as a result.*

In this month's threads, I start *with a discussion of the proper way* to *measure an RS-422 line. Since it's a differential signal, it's not as easy as touching a* single scope probe *to* the *line.*

Next, we look at decoding a low-speed datastream coming through a trunked *radio system. There is more to this thread than would fit within these pages, so if it sparks your interest, give the BBS a call and read the whole thing.*

Finally, it's time to cut some foam with a heated wire, but how do you design the drive electronics for such a wire?

Measuring RS-422 signals

Msg#:12291
From: Dan Walker To: All Users

What is the proper way to measure RS-422 if you want to check the amplitude and the condition of the waveform? I have been measuring from the positive terminal to ground with a Fluke Scopemeter. I have heard some people say you should measure from the positive terminal to the negative terminal with the scope.

Msg#:12926
From: James Meyer To: Dan Walker

I'd measure first one side with respect to ground and then the other. In most cases, they will be mirror images of each other. If they aren't, then something's not quite right somewhere.

If you take only *one* measurement, either side to ground or differentially between sides, you could miss something important.

Msg#:13133
From: John Wettroth To: Dan Walker

Gosh, you've got an isolated scope-just measure from positive to negative. It is a differential standard and looks at the difference between positive and negative. The idea is to reject all the common-mode crud that is present on both wires. Your Fluke Scopemeter is the ideal instrument to make these types of measurements. If you measure one wire or one wire at a time, you really can't tell anything unless there is no common-mode noise and there is a path to ground somewhere. Theoretically, the signals have no relation to ground, in practice there is a ±7-volt common-mode limit.

Msg#:17242
From: Pellervo Kaskinen To: Dan Walker

As is so often, it depends.. .

If you have limited facilities, you measure what you can. If that does not appear to adequately cover your information needs about the waveforms, you take the next more complicated approach.

I personally prefer always making two measurements on the differential signals, maybe three. The two measurements can be any combination of the actual difference signal and one polarity versus common or the two signals against common at the same time. In the last case, I can mentally process the difference.

But most of the time it is just so simple to turn the two channels of the scope into a quasidifferential mode. If I have the two signal lines attached and a difference displayed, then I can turn one input selector to grounded position and I see the individual signal of the other line. I can flip this over as many times as I want with minimal effort.

The reason I want to see the individual signal(s) is that there can be a common-mode level in excess of the receiver or transmitter capability. The differential display may or may not reveal that possibility. On the other hand, it is the differential signal that is supposed to carry the information. If I do not measure that, I'm assuming too much. 8-)

Low-speed data

Msg#:17234
From: Ben Stedman To: All Users

I'm looking for suggestions on how to decode some low-speed data that's used in a trunked radio system. The data is used to steer mobile radios to the proper repeater and

to identify them to the system to provide a small measure of privacy. The particulars are:

Bit rate: 300 bps
Message length: 40 bits
Repetition rate: continuous

As far as I know, there is no framing on the individual parts of the message, however there is a sync pattern consisting of 101011000 (9 bits) at the start of each message.

The entire message looks like this:

9 bits sync
1 bit area
5 bits goto repeater
5 bits home repeater
8 bits id code
5 bits free repeater
7 bits checksum

Should I monitor the incoming bits until a match is found with the sync pattern and then save the next 31 bits? Or should I read 40 bits into a buffer and then scan the buffer for the sync pattern? Or is there some other good method?

I plan on using a PIC or an 803 1 to do the processing and then display the information an a LCD. Thanks in advance for any ideas and comments.

Msg#:17252
From: Russ Reiss To: Ben Stedman

As they say, Ben, there are many ways to skin a cat. Any method that gets the job done is usually OK, but it would seem to me that it is simpler to just keep monitoring the input until you see the sync code, then grab the following 3 1 bits and use them. Otherwise, you need to feed everything into a circular buffer, mark the location of the sync (when you find it), and keep everything else in sync with that position. It certainly can be done, but sounds more complex than the first approach, and I can't see what's gained by it. Just be sure that the sync pattern is truly unique and cannot appear as some combination of data in other fields (presumably the designer of the encoding technique thought of this!) :}

Sounds like an ideal project for a PIC chip. You might check out my June '94 article in ***INK*** for how simple development would be with a PIC16C84. Their EEPROM capability makes them ideal for "interactive" development of a project like this. Seldom does the code work right the first time, and once you get it running, you always think of new things to add. You typically end up unplugging, UV erasing, and reinstalling many, many chips in this process. That's why I so much enjoy onboard reprogrammable micros like the '84.

Msg#:19588
From: James Meyer To: Ben Stedman

I would start out by putting the pulse string into my desktop PC first. A little bit of fiddling around with various methods of detection there would be *much* easier than the compile.. .burn...crash...recompile...erase...burn...crash method of starting out directly on the microcontroller.

Either that, or I would use a simulator program for the target micro that I could run on the PC.

Meyer's Maxim #42: "A peek at the answer is worth a thousand guesses."

At least it worked for me during high school.. :-}

Msg#:21006
From: Dave Tweed To: Ben Stedman

What you have is called a "framed" data stream, and the Y-bit pattern is called the "frame pattern." Finding and maintaining frame alignment in a potentially noisy (i.e., full of bit errors) channel is nontrivial, but not terribly difficult, either.

First of all, you cannot assume that the frame pattern will not appear elsewhere in the data stream. The only way to confirm that you have the correct frame alignment is to check that it repeats at the expected 40-bit frame period. False frame patterns will not.

The general technique is to have two states in the software: in-frame and out-of-frame. When you are out-of-frame, you search the incoming data bit-by-bit until you find a valid frame pattern, then go in-frame. When you are in-frame, you split out the individual data fields, and then check for another frame pattern.

Here's where things get tricky-if you don't find another frame pattern, you do not necessarily want to go out-of-frame immediately. You may want to see if only one or two bits are wrong and stay in-frame if so. You may simply want to wait for another frame period and only go out of frame if you fail to see the frame pattern two or three times in a row.

Note that you do not necessarily need to buffer the incoming data; you can do all of this on the fly as the bits come in.

By the way, 300 bps sounds easy-the systems I build normally do this sort of thing at I.544 Mbps and up. I also have some experience with trunking radio systems.

Also, the 7-bit "checksum" on a 40-bit message sounds more like an error-correcting code-you can use it both to verify frame alignment and to greatly improve the overall reliability of the data. In a *really* noisy environment, you could even go so far as to apply the error-correction algo-

rithm at every possible frame alignment when out-of-frame. In this case, you'd need the 40-bit buffer.

Msg#:21608
From: Ben Stedman To: Dave Tweed

Thanks for the info concerning framed data streams. I think it will be relatively straightforward to code the technique you suggest.

As far as the 7-bit error-correcting code at the end of each packet goes, I have no definitive information as to how it is calculated, but since this is an "educational experience" project, I can just ignore it for now.

However, the next step is to attempt to *encode* this information as well, and I guess I'll need to study those last 7 bits carefully.

Msg#:23757
From: Michael Millard To: Ben Stedman

From your description and my knowledge of trunking formats, it looks like you are describing an EF Johnson LTR format. This comes in Uniden, Kenwood, Trident, Zetron, and other flavors, but for the sake of backward compatibility, none change the actual repeater data bus. At least not at the tower site location. Mobile formats may differ slightly depending upon signaling options.

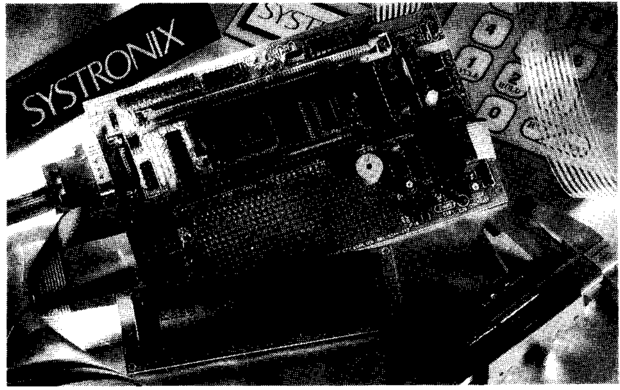
LTR is an open-system architecture. You can save yourself a lot of headache by just asking for the specification from the respective manufacturers or see the EF Johnson Trunking System Specification Literature.

Wire heating

Msg#:34799
From: Andrew Dignan To: All Users

I am trying to find a way to keep a wire at a constant temperature. The wire is cutting through a foam insulating material, the ends being a foam core wing for aircraft. At present I am using an AC variable transformer to control the temperature but am looking for something a little more controlled and automatic. I would guess that a constant-current power supply would be the answer? Are there circuits out there that can handle this? The power demands

NEW! UNIVERSAL DALLAS DEVELOPMENT SYSTEM from $199!



- It's a complete 8051-family single board computer!
- One board accommodates any 40 DIP DS5000, 40 SIMM DS2250, 40 SIMM DS2252, or 72 SIMM DS2251, 8051 superset processor! Snap one out, snap another in.
- Programs via PC serial port. Program lock & encrypt.
- LCD interface, keypad decoder, RS232 serial port, 8-bit ADC, four relay driver outputs, four buffered inputs.
- Power with 5VDC regulated or 6-13 VDC unregulated
- Large prototyping area, processor pins routed to headers
- Optional enclosures, keypads, LCDs, everything you need
- BCI51 Pro BASIC Compiler w/50+ Dallas keywords $399

SYSTRONIX® TEL: 801.534.1017 FAX: 801.534.1019
555 South 300 East, Salt Lake City, UT, USA 84111

#136

are from 50 to 100 watts. Any suggestions? Thanks ahead of time.

Msg#:34824
From: James **Keenan** To: Andrew **Dignan**

I have made my own foam cutter using a variable transformer and a transformer salvaged from a microwave oven. The variable transformer feeds the primary side of the microwave transformer. The microwave transformer has had the secondary winding removed and replaced with 12 wraps of 10-AWG insulated wire. This whole mess heats a 30″ length of 0.032" wire made of 304 stainless steel (Acft. safety wire). This circuit is not automatic, but it works very well and keeps the "*zap*" factor in the safe range. Experiment with the number of wraps around the microwave transformer so you have the desired temperature of the wire at midrange on the variable transformer.

I think a constant-current power supply will not do you any good because it has no control of how fast the wire loses heat.

Did you consider some sort of motorized drive for your wire bow to achieve a constant cutting rate?

Msg#:35476
From: Andrew **Dignan** To: James **Keenan**

James, thanks for the input. I do have something in mind for controlling the bow. I have written a program that generates airfoils on the screen. I am working on creating a stepper motor drive. It involves driving two x-y axis tables at each end of the wing. I am doing it for the fun of it! You can spend $2000 to $3000 and get a system to do this, but you can't say, "I built that."

As for the wire heating, there is a box out there that is made for this purpose. It will hold the temperature of a wire fairly constant (with in 5"). I don't know what they are doing to get this done, so the challenge continues.. .

Msg#:34928
From: Lee **Stoller** To: Andrew **Dignan**

If you truly want to keep the wire temperature constant, you must design a circuit that feeds current to the wire such that the wire's *resistance* stays constant. This means you must measure both the current through the wire and the voltage across it, and juggle things to keep the ratio constant.

Your best bet is probably to use AC to power the wire, and maybe an SCR or triac to control the current. The difficulty that you face is the need to find sensors that will respond to the true RMS values of the voltage across the wire and the current through it. Then you need a system (maybe a microcomputer?) that will calculate the product of the two measurements. Only then are you in a position to change the thyristor's firing angle to compensate for changes. This is a standard, but not trivial, control system problem. You may need the full PID treatment for satisfactory operation.

Msg#:36731
Dignan To: **Stoller**

I

Msg#:37282
Dignan

Msg#:42935
Dignan

constant-temperature-

you should see the reason for that choice.

CONNECTIME

Of course, different materials and different cutting speeds may cause a need to adjust the base line (the supply voltage). But as Lee mentioned, you would only need a simple variac to feed the primary of the main transformer. Or you could try to use some triac circuits for the same because of price concerns.

A triac circuit has a bad tendency of pumping DC through the transformer. Most "sloppy" transformers can adapt to it, but some "designed to the limit" transformers might develop severe convulsions.

Speaking of DC through a transformer, here is a story I just heard and explained to the people who told it to me.

It appears there was a defective welding power source that blewup the utility transformer next to the customer's plant. The local distributors sent it back to the manufacturer for repair. In due time it came back and was delivered to the customer, installed and tested. In 10 minutes, there was a big bang outside and all the lights went out. The utility transformer on the pole had more or less exploded. No fuses or circuit breakers inside the building had opened. What was the cause?

In my theory, the primary rectifier of the switching power supply was defective. One leg of the full wave rectifier was open. The unit started pumping DC through the AC circuit. Since there was no main transformer inside the welder before the inverter, no local saturation took place. Also, the RMS current through the loop was not too high to open any fuses or breakers. But the utility transformer core saturated and its primary current became enormous. BANG!

I don't mention the brand of the welder, as this kind of thing can happen to anybody. Luckily it was not us, though!

Msg#:44145
From: Lee Stoller To: Pellervo Kaskinen

I can't resist breaking in here to mention something I saw in an old GE manual: a neat way of varying the AC voltage to a transformer without any of the DC problems that you mentioned. You put the primary of the step-down transformer in series with the primary of the transformer to be controlled. The secondary feeds a bridge rectifier, which is connected to the collector and emitter of a power transistor. This provides a nice, simple, and isolated connection (the BE junction) for your control electronics. By varying the base current, you vary the equivalent resistance of the transformer primary in series with the load. Neat?

We invite you to 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, 2400, 9600, or 14.4k bps. For information on obtaining article software through the Internet, send E-mail to info@circellar.com.

ARTICLE SOFTWARE

Software for the articles in this and past issues of *Circuit Cellar INK* may be downloaded from the Circuit Cellar BBS free of charge. For those unable to download files, the software is also available on one 360 KB IBM PC-format disk for only **$12.**

To order Software on Disk, send check or money order to: Circuit Cellar INK, Software On Disk, P.O. Box 772, Vernon, CT 06066, or use your Visa or Mastercard and call (203) 875-2199. Be sure to specify the issue number of each disk you order. Please add $3 for shipping outside the U.S.

437 Very Useful 438 Moderately Useful 439 Not Useful

80C52-BASIC CHIPS IN VOLUME

Micromint's 80C52-BASIC chip is an upgraded replacement for the venerable Intel 8052AH-BASIC chip. Ours is designed for industrial use and operates over the entire industrial temperature range (-40°C to +85°C). Available in 40-pin DIP or PLCC.

80C52-BASIC chip $19.00

OEM 100 *qty.* $12.00

BASIC-52 prog. manual $15.00

Call (203) 871-6170 or 1-800-635-3355 to order-

MICROMINT, INC. 4 PARK STREET, VERNON, CT 06066

#119

STEVE'S OWN INK

One of Those Days

After all these years, I think I'm finally losing it.

This morning, I got ready for work and forgot breakfast on the way out. I had to return to the house to get the truck keys. But, of course, the alarm was already triggered and, when I pressed the garage-door button. I nearly pulled it off the tracks because it was still locked. As I skidded out the driveway from what must be the only still-snow-covered spot in the whole state, I blasted by a neighbor and nearly suffocated him in a cloud of sand.

Ordinarily, I don't have to be any place in particular. but I knew Ken would be looking for me. Guilt about my long overdue editorial swirled in my head as I pulled into the Xtra Mart for coffee. I was so distracted that I hardly noticed the little old lady who thought I wanted to play chicken for the one remaining parking place. She nearly became my new hood ornament.

Getting the self-serve coffee was my next experience. As an engineer, I usually approach even that in a logical manner. I put the sugar in the cup, I put the cream on top of that, and then I add the coffee. which mixes everything. I have to grit my teeth as I watch others whip up water spouts, which slop coffee all over the place. This morning the cream container needed refilling, the first cup I picked leaked, they had to search for more of the right-size covers, and somehow I inadvertently got Columbian Raspberry Walnut-something coffee. Ugh.

When I finally got the coffee, I got in line behind someone whose idea of breakfast was an extra-large, red-hot, beef and bean burrito, two hot dogs with chili and sauerkraut, three chocolate-covered doughnuts with sprinkles, and a quart of Coke. The fragrances wafting from the guy in line behind me convinced me that turning to see his morning selection would only add insult to injury.

The two mile trip to the office was mostly uneventful, but arrival presented yet another problem. I'm generally a nice guy and generous to a fault, The one minor, insignificant, frivolous, negligible, trivial event that really frosts my cookies, however, is pulling into the parking lot and finding my parking space occupied. The mere fact that a person ignores not one, but two sets of signs announcing a variety of dire consequences if they park there only suggests that a challenge has been advanced.

Do I call a wrecker and have this bloke unceremoniously dragged off on a hook? Do I push a few levers and leave tire tracks across the guy's roof? Or, do I resort to vigilante tactics?

To my knowledge, I'm the only guy in Connecticut with a permanently mounted, 8000-lb. winch on the front of his 4-wheeler. To date, the only time I used the winch was the last time someone parked in my spot. I reeled out the cable, looped it around the bumper hitch, and, with a grin that only a Cheshire cat could appreciate, I pressed the control button and bodily removed the offender.

These past thoughts flashed through my mind as I swung into the parking lot. Would this be a personal or professional tow?

Unfortunately, this morning I had neither the will nor the endurance for yet another crisis. Instead, I instantly redirected my aim and made a perfect 4-point slide into the General Managers parking spot. Knowing her, she'd have the guy's car ground into little pieces.

As I opened the building door, I felt I was finally in a place of safety. With a properly placed "Meeting in Progress" sign on my office door I might yet resurrect the day. That was, until I entered the hallway and came face to face with Ken, "You're going to have an editorial for me today. Right, Steve?"