EMBEDDED PC MONTHLY SECTION

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#96 JULY 1998

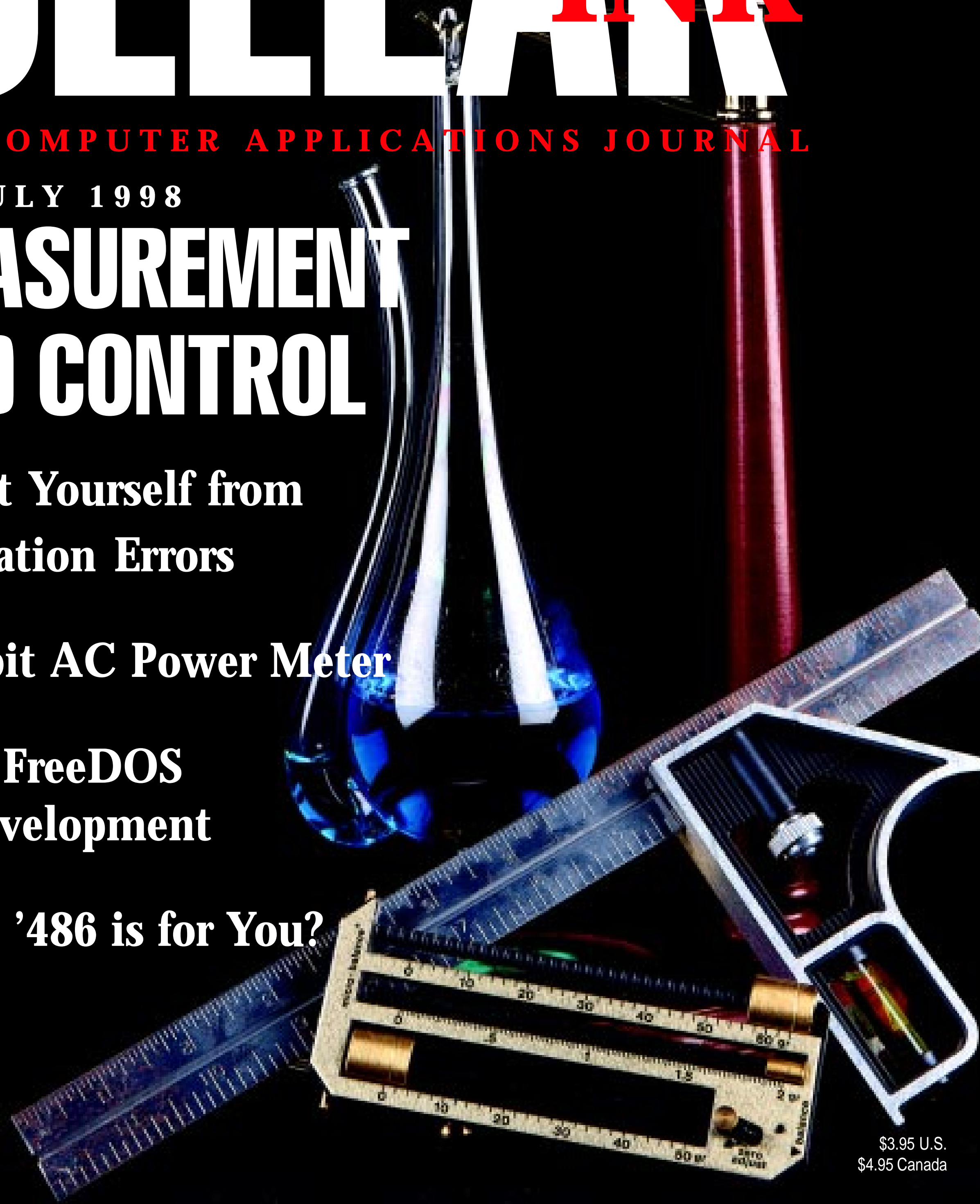
MEASUREMENT AND CONTROL

Protect Yourself from Calibration Errors

An 8-bit AC Power Meter

Using FreeDOS for Development

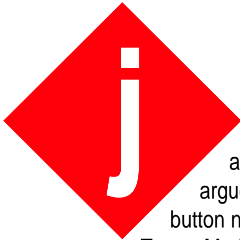
Which '486 is for You?



$3.95 U.S.
$4.95 Canada

TASK MANAGER

Professional Quirks



Just this week, Elizabeth and I were laughing about how eccentric editors are. We do things like argue about commas, apostrophes, and whether the button name on a GUI should be in Letter Gothic or Trump Mediaeval fonts. Essentially, it seemed to both of us that editing is the sole profession that encourages perfectionism, obsessive-compulsive behavior, and hypersensitivity. In other words, to be a good editor, you have to care a *lot* about a *lot* of little things.

Similarly, engineering encourages some rather unusual idiosyncrasies. Just take a meander to your coffee room—no, I stand corrected—your Coke machine. If two engineers happen upon the poor unsuspecting machine at the same time, what do they talk about? You've got it: how many milliseconds it takes to execute 70,000 lines with 64 KB of memory. Never mind that they can't do anything in milliseconds, the computer can. But, in fact, that's not good enough, so let's talk microseconds. When can we advance to picos? Engineers get their Coke because their discussions of faster and faster speed with yet less power have the poor machine trembling in fear. It imagines itself spewing cans at a rate of 17,000,000 per picosecond if the engineers but wish it.

The engineer's penchant for control gets even more exaggerated when you move into software. There you have an inflexible grammar that refuses to cooperate at all unless your variables are referenced exactly and your punctuation is just so. And, should there be any question about what is intended, just look to the top of the code. Everything is defined quite explicitly. They even tell you what library they got their source from.

When you look at all the oddities of engineers and editors, it's a wonder you get any creativity out of either professional. But, there you have it: one of the Seven Wonders of the World. You weigh power and time and make wonderfully useful widgets, while I weigh grammar and sentence flow and make people laugh or think.

This issue is a perfect example of what I'm talking about. The issue is about measurement and control, but in nearly all of the articles, the engineer author applies these concepts specifically to a design or problem that they're fighting with. Take Mike Smith's article. In it, Mike shows how storing decimal numbers as binary can get you into some very messy calibration errors. Rick May builds a power meter that gives him feedback about where the inordinate load comes from. Carol Hovenga Fancher starts a two-part series on smart cards, while Dan Cross-Cole uses an ADC board in conjunction with his PC to measure and fix radon levels in his basement.

In *EPC*, Pascal Dornier finishes his survey of '486 technology (he promises an overview on Pentiums at a later date). Ingo applies networked communication to RTOS applications, and Fred wraps up his virtual-tools series with a real-world application.

Pat Villani completes his MicroSeries on FreeDOS by running us through an airline ticketing application, Jeff applies a little assembly language to Steve and his alarm monitor, and Tom looks at a new 8-bit wonder that's selling for less than 50¢.

And there you have it—a lot of creativity, all focusing on measurement and control.

Janice

janice.hughes@circuitcellar.com

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

EDITOR-IN-CHIEF
Ken Davidson

MANAGING EDITOR
Janice Hughes

TECHNICAL EDITOR
Elizabeth Laurençot

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Ingo Cyliax
Fred Eady
Rick Lehrbaum

NEW PRODUCTS EDITOR
Harv Weiner

ASSOCIATE PUBLISHER
Sue (Hodge) Skolnick

CIRCULATION MANAGER
Rose Mansella

BUSINESS MANAGER
Jeannette Walters

ART DIRECTOR
KC Zienka

ENGINEERING STAFF
Jeff Bachiochi

PRODUCTION STAFF
John Gorsky
James Soussounis

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

ADVERTISING

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

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

CONTACTING CIRCUIT CELLAR INK

SUBSCRIPTIONS:
INFORMATION: www.circuitcellar.com or subscribe@circuitcellar.com
TO SUBSCRIBE: (800) 269-6301 or via our editorial offices: (860) 875-2199

GENERAL INFORMATION:
TELEPHONE: (860) 875-2199 FAX: (860) 871-0411
INTERNET: info@circuitcellar.com, editor@circuitcellar.com, or www.circuitcellar.com
EDITORIAL OFFICES: Editor, Circuit Cellar INK, 4 Park St., Vernon, CT 06066

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

For information on authorized reprints of articles, contact Jeannette Walters (860) 875-2199.

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

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 © 1998 by Circuit Cellar Incorporated. All rights reserved. *Circuit Cellar INK* is a registered trademark of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

12 **Unplanned Calibration Errors in Embedded Systems**
Mike Smith

22 **A PIC-Based AC Power Meter**
Rick May

28 **Designing for Smart Cards**
Part 1: What's a Smart Card All About?
Carol Hovenga Fanchen

58 **Using a PC for Radiation Detection**
Modifications for Multichannel Analyzer Capability
Dan Cross-Cole

66 **MicroSeries**
FreeDOS and the Embedded Developer
Part 2: Using the Kernel
Pat Villani

74 **From the Bench**
An Alarming Improvement
Part 2: Assembly Language Takes the Race
Jeff Bachiochi

80 **Silicon Update**
The Micro Price is Right
Tom Cantrell

Task Manager 2
Janice Hughes
Professional Quirks

Reader I/O 6

New Product News 8
edited by Harv Weiner

Advertiser's Index/ August Preview 95

Priority Interrupt 96
Steve Ciarcia
It's All in How It's Done

INSIDE ISSUE 96

EMBEDDED PC

36 **Nouveau PC**
edited by Harv Weiner

40 **'x86 Processor Survey**
Part 2: '486-Class Embedded CPUs
Pascal Dornier

45 RPC **Real-Time PC**
Network Communication
Ingo Cyliax

52 APC **Applied PCs**
A New View
Part 3: Sensors and Measurement Tools
Fred Eady

www.circuitcellar.com
★ July's Password: Power ★

READER I/O

ADA—NOT AS BAD AS YOU THINK

"Software Development for RTOSs" (*INK* 93) states that Ada is "sometimes difficult to use for expressing real-time issues." It's certainly true that some real-time and concurrency issues are hard to deal with—in any language.

Burns and Wellings' book *Concurrency in Ada* discusses many of the issues as well as the solutions developed over a decade's experience with Ada in a large number of real-time systems. Even those unfamiliar with Ada may find that some of their problems have already been faced, and better yet, working solutions are available.

Tom Moran
tmoran@bix.com

YOU'RE BANG ON FOR DESIGNERS

The Microchip rep dropped by last week and left me a large check. I just wanted to say thanks for holding the Design98 contest.

It is good having a journal in our industry that focuses on accomplishment and real-world problem solving rather than the philosophy of embedded systems and C++ as most of the others do!

Hank Wallace
hank@aqdi.com

IT'S NO TYPO

With respect to Figure 1 in "8*x*51 EPROM/Flash Microcontroller Programmer" (*INK* 93, p. 19), here's some new information: for chip U1, use a DS275 not a DS1275. I thought the DS1275 looked interesting for a project I'm working on, but I couldn't find it on the Dallas Web site. DS275 looked like the right part number, and although at first I thought it was a typo, I soon realized it wasn't.

The DS1275 came out around 1992 as a line-powered RS-232 transceiver chip, and that is what's used in the circuit in the article. However, around 1996, Dallas replaced the DS1275 with the DS275. There is no mention of the DS1275 in the DS275 datasheet, nor is any information to be found any longer about the DS1275.

The only difference I can see on the intro page of the two datasheets is the older part conforms to RS-232-C while the newer does RS-232-E. I haven't done extensive comparisons, but I am quite sure that you may use the DS275 in place of the DS1275.

Tom Riggs
thriggs@netusal.net

NEVER SAY DIE

I must tell you, punched tape is alive and well in the CNC machine tool industry, contrary to what I read in "Interfaces and GUI-Building Packages" (*INK* 89).

Magnetic media is occasionally used on the manufacturing floor, but polyester (Mylar) punched tape is usually preferred. If you were to drop a reel of punched tape into a drum of hot machine oil, leave it there for a week, and take it out, it could be unreeled onto a clean surface, patted dry, and read in any punched-tape reader. Try doing that with a floppy diskette.

Robert Michaels
robert.michaels@online.sme.org

THE PROBLEMS WITH SUPPLIERS...

I enjoyed "What You Get with a Handshake" (*INK* 93) and would like to make some comments on our relationships with distributors versus catalog sales.

The greatest value of using Digi-Key as a catalog sales company is that we can order nearly everything to build a prototype and they usually have everything in stock. Digi-Key is great for one-stop shopping, and the catalog is useful for getting worst-case pricing for feasibility studies. However, prices are a lot higher than what we can negotiate with distributors.

With distributors, buying the first order is always an adventure, going back and forth, playing them against each other to get the best price. Afterward, the intensity is much lower, usually consisting of asking, "Can you come down a dollar?"

And, why is there the chronic shortage of Maxim and LTC parts? We love them from an engineering standpoint, but production has to inventory at least four months' worth to ensure we can ship products.

Jim Stewart
jstewart@jkmicro.com

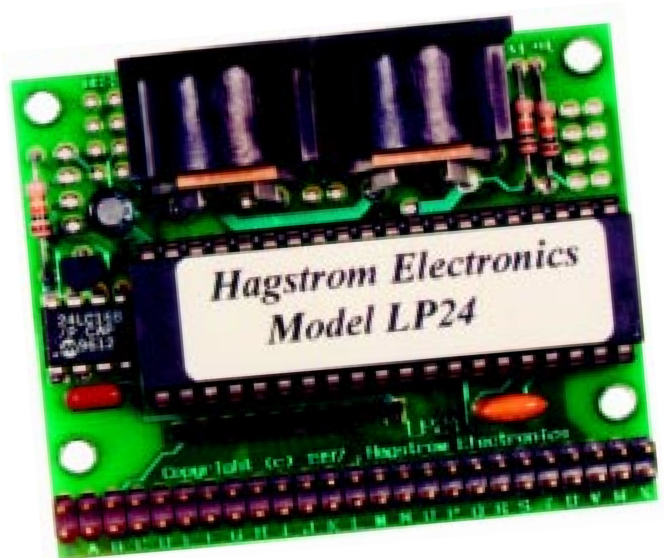
NEW PRODUCT NEWS

Edited by Harv Weiner

PROGRAMMABLE KEYBOARD EMULATOR

The **LP24** programmable keyboard emulator interfaces keypads and switches to the keyboard input of a PC ('286 and higher). Each switch input may be programmed to emulate any of the standard keys found on a 104-key keyboard. The LP24 I/O lines may be programmed as columns or rows for scanning up to a 12 × 12 matrix. This programmability provides easy interface to existing keypads or switches. The LP24 also lets the user execute multiple keystrokes (e.g., Ctrl-F1) from a single input. This small (2″ × 2.5″) unit emulates all features of a PC keyboard, so it may be used alone or simultaneously with an existing PC keyboard.

The LP24 is priced at **$59.95** in quantities of 100, which includes programming software.



Hagstrom Electronics
(888) 690-9080 • (607) 786-7523
Fax: (607) 786-5190
www.hagstromelectronics.com

MULTIPLE-PROCESSOR DSP

The **HECPCI-1** is a 3U CompactPCI board that supports up to four Texas Instruments TMS320C4*x* processors. Multiple HECPCI-1 boards can be connected together to form a DSP network with unlimited processing, data-acquisition, and I/O capability.

The 3U CompactPCI board configuration has a single slot for a **TIM-40 module**. This modular architecture enables a custom system to be put together to suit a particular application, using entirely commercial-off-the-shelf components. A variety of plug-in TIM-40 modules for use with the HECPCI-1, including multiprocessor DSP, image-processing, data-acquisition, and communications interfaces are available.

There are six buffered TMS320C4*x* Communication Ports (Comports) on the HECPCI-1 for communicating with other TMS320C4*x* hardware, so you can construct a network of any size. These Comports can be used to connect multiple HECPCI-1 CompactPCI boards or to connect to other system hardware such as PCI, ISA, PC/104, VME, SBus, and custom expansion boards.

The HECPCI-1 communicates with a host PC through the PCI interface, with two independent communication channels (one of which can master PCI transfers). It includes a JTAG controller to provide support for standard debugging software, like GO-DSP's Code Composer.

A PC API provides transparent software support in a consistent, structured, and user-friendly manner to ease the use of DSP hardware from DOS, Windows 95, and Windows NT. A variety of third-party code-generation tools, loaders, run-time environments, function libraries, and debugging tools are also available.

The HECPCI-1 sells for **$2050**, and pricing for TIM-40 modules starts at **$2575**.

Traquair Data Systems, Inc.
(607) 266-6000
Fax: (607) 266-8221
www.traquair.com

NEW PRODUCT NEWS

DATA-ACQUISITION SYSTEM

Featuring eight independent 24-bit ADCs, the **PAR24B** data-acquisition system achieves 22-bit true single-sample accuracy at a 20-Hz sampling rate, with a maximum sampling rate of 1 kHz. With eight individual A/D subsystems, there are no cross-talk or settling problems as is common with multiplexed systems.

The system is interfaced with standard bidirectional PC parallel ports, making it ideal for laptop or desktop machines when installing ISA or PCI bus cards is difficult or impossible. Optionally, the parallel port can be run in EPP mode for fast data transfers. Parallel port cable lengths up to 30′ can be used to run the system separated from the PC, if necessary for noise or other conditions.

Overall analog input gain may be set either by jumper-selectable input resistor dividers or software. All eight channels have differential inputs for maximum noise rejection. Once converted, incoming digitized data is fully buffered with an onboard FIFO, enabling continuous data acquisition even during heavy interrupts or multitasking. Typical FIFO buffer depth is 8.5 s at a 10-Hz sampling rate, which provides ample time for task switching under Windows 95 to save the acquired data.

Complete software support is supplied, including drivers for DOS, Windows 95, and National Instruments' LabVIEW. Finished application programs include `SCOPE24B.EXE`, a full GUI acquisition kernel displaying acquired data as horizontal traces on screen, as well as a full LabVIEW application displaying data from each channel as a digital display. Extensive documentation including circuit diagrams and source code is included with every system.

The PAR24B system sells for **$800**.

Symmetric Research
(702) 341-9325
Fax: (702) 341-9326
www.symres.com

THERMOCOUPLE MEASUREMENT SYSTEM

The **EZ-View-TA** thermocouple measurement and data-acquisition system attaches to a standard printer port of a laptop or desktop computer to provide an affordable approach to portable thermocouple data acquisition. The included software self-installs, and there are no addresses to set or switches to change. EZ-View-TA is compatible with all MS-DOS 3.0+, Windows 3.0+, and Windows 95–based computers with VGA or better screens. Operational modes include monitoring (oscilloscope mode) and data acquisition (record mode).

The EZ-View-TA features six simultaneous temperature channels, as well as letting you mix or match any common thermocouple (B, E, J, K, N, R, S, T) gain adjustments. It also offers bias offsets, scale selection, sampling rate and run-time selection, channel labeling, triggering, autoscaling, and remote-start options. Data is transportable to standard spreadsheets and can be zoomed for detailed analysis. A unique notes feature enables you to attach a brief text description of the data to each saved file.

The EZ-View-TA system ships complete with the data-acquisition module, power supply, data cable, and complete instruction manual for only **$399**. Options include 16-bit data resolution, remote battery power supply, and thermocouple sensors.

Mid-Atlantic Systems Co.
(810) 750-4140 • Fax: (810) 629-4988
www.mid-atl-sys.com

NEW PRODUCT NEWS

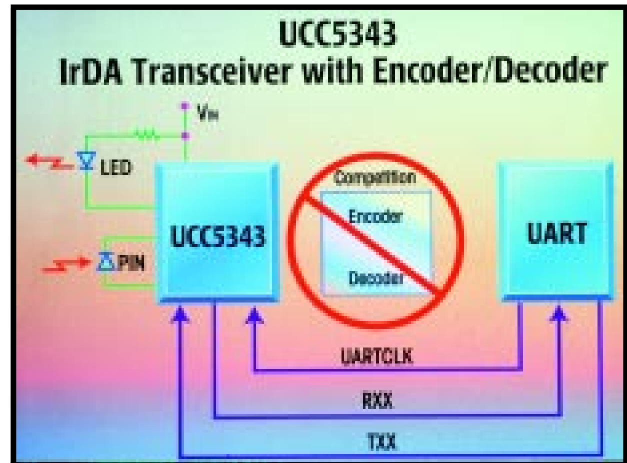
IrDA TRANSCEIVER

The **UCC5343** IrDA transceiver with encoder/decoder provides data format translation between a standard UART and IrDA to minimize the need for external devices. The chip supports the Physical Layer specifications of the IrDA 1.0 standard. Applications include wireless communication for portable devices like pagers, PDAs, cell phones, and hand-held computers.

The UCC5343 is readily interfaced directly to a standard UART. A limiting transresistance amplifier detects a current signal from a PIN diode and drives RXX pulses into a UART. Wide dynamic range enables the receiver to detect input currents from 200 nA to 50 mA.

The receiver signal path is frequency limited by an internal band-pass filter to reduce interference from other IR energy sources. Receiver output is designed for direct interface to standard UARTs and Super I/O devices at data rates up to 115.2 kbps. Internal resistors for decoupling the PIN diode supply minimize external components.

The UCC5343 has low current consumption in the active mode. The transmitter section has a low-impedance totem pole MOSFET output capable of sinking 300 mA from an output LED at 3 V, and 500 mA at 5 V.



The UCC5343N is priced at **$4.35** in 1000-piece quantities.

Unitrode Corp.
(603) 424-2410
Fax (603) 424-3460
www.unitrode.com

NEW PRODUCT NEWS

DIGITIZER/AVERAGER BOARD

The **TD-250** digitizer/averager board from TerraData, a division of DAS, features a sampling rate of 250 MHz and a maximum analog bandwidth of 125 MHz. This board is designed to operate in an IBM PC–compatible computer and occupies one 16-bit ISA slot.

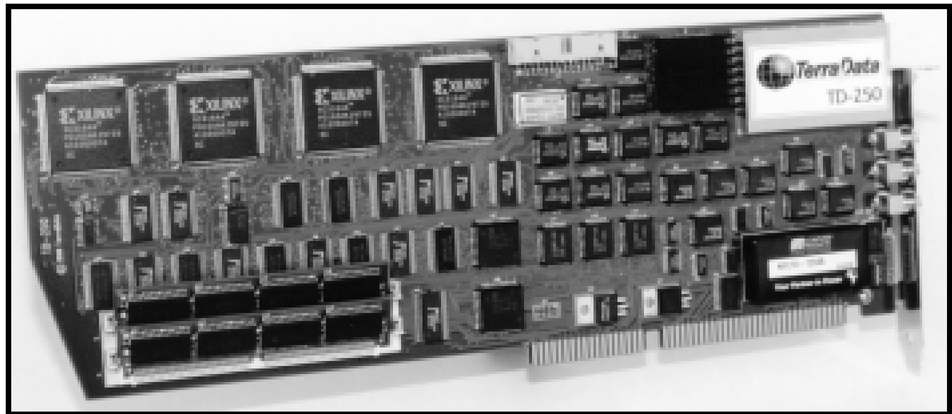
A unique, real-time hardware averaging section increases its effective resolution from the native 8 bits up to 11 bits, for repetitive signals. The number of samples is software selectable and can be set in binary steps over the range of 1 to 32,768.

The TD-250's 1 MB of onboard SRAM is organized as an array of 512-KB samples. In addition to the onboard memory, the TD-250 is capable of expanding its memory capacity via DT-Connect as well as additional memory-expansion boards. The DC-Connect option also offers 20-MBps throughput.

The TD-250 operates in one of three modes—burst, average, and powerdown. In burst mode, the entire SRAM buffer is filled at the sampling frequency. In average mode, 32-KB samples of averaged data are stored in SRAM. The power-down mode is used when the TD-250 is not acquiring data, and it helps to reduce heat buildup.

The TD-250 includes real-time digital scope software that provides access to its hardware setups. LabVIEW drivers will be available as well.

TerraData
(801) 224-8080
Fax: (801) 224-8087
info@dasengr.com



COLOR CCD CAMERA

The Hitachi **KP-D8** color CCD camera is one of the smallest CCD cameras available with DSP and remote computer control capability.

The self-contained camera measures 22 mm × 22 mm × 86 mm and offers optional remote control via an RS-232 port. Because of its size, excellent image quality, and remote accessibility, the KP-D8 is well suited for medical applications, specialized monitoring or process control in factories, or undercover surveillance situations.

The KP-D8's image quality is further enhanced with DSP technology. DSP gives the camera an automatic 2H enhancement, automatic aperture correction, backlight correction, and three white balance modes (autotracking white, memory, and manual adjustment of red and blue gains and black balance).

In the NTSC version, the KP-D8 has a total of 410,000 pixels with a minimum sensitivity of 4 lux at F1.2. The PAL version has 470,000 pixels per frame with a minimum sensitivity of 4 lux at F1.2. The KP-D8 has 470 TV lines of resolution.

A ⅓″ CCD with a microlens increases the camera's light sensitivity. This camera has an auto-electronic shutter and provides composite and Y/C outputs. A lens and DC power source are all you need to make the camera operational.

The KP-D8 sells for **$1150**.

Hitachi Denshi America, Ltd.
(516) 921-7200
Fax: (516) 496-3718
www.hdal.com

FEATURES

12 Unplanned Calibration Errors in Embedded Systems

22 A PIC-Based AC Power Meter

28 Designing for Smart Cards

58 Using a PC for Radiation Detection

Unplanned Calibration Errors in Embedded Systems

FEATURE ARTICLE

Mike Smith

If bugs in your system cause damage, who's liable? Mike puts you on the witness stand to uncover the problems and how they got there. Fortunately, he also shows an easy-to-implement fix that puts you in the clear.

general interest in program bugs is obviously building when a noncomputer magazine features an article on embedded systems. This particular article was about the millennium bug. At first glance, this is "just an industry problem" and all that Joe and Jane Public need do is bunker down in a warm house with a bottle of wine and toast each other into New Year's 2000.

However, although many accounting systems are close to year-2000 compliance, *The Economist* (October 4, 1997) questions, "Just how many embedded systems are compliant?" You know, those little embedded controllers that let the gas and electricity flow down the utility pipelines into your house? In Canada, where I am, this is rather important—New Year's comes in cold midwinter!

The millennium problem is one of the many hidden defects that can be created by well-intentioned programmers of embedded systems. Using two digits to represent the year was a sensible solution to a known memory limitation of the '60s and '70s.

Unfortunately, unforeseen side effects occur with many sensible solutions. So, follow along to determine whether or not your code might in-

Listing 1—*This embedded-processor program reads the temperature from an eight-bit sensor and then corrects and logs temperatures in the range of 32°F to 154.5°F. A warning is issued if the temperature falls below 40°F.*

```
#define CORRECTION (4 * 2)          // Temperature correction needed
#define WARNING_LEVEL (40 - 32) * 2 // Operate warning at 40°F
#define DEVICE_ADDRESS 0x500000     // Memory-mapped device location

// Set device pointer
unsigned char *pt = (unsigned char *) DEVICE_ADDRESS;
unsigned char value;

for ( ; ; ){
  Wait_5_minutes();
  value = *pt;                      // Read temperature
  value = value - CORRECTION;       // Correct temperature
  Log_Temperature(value);           // Log temperature
  if (value <= WARNING_LEVEL)       // Issue warning
    Activate_Siren();
}
```

clude some unexpected defects. If your code is valid, then what about the ROM code in the heart monitor when you suffer from a heart attack caused by work-related stress?

DOWN SOUTH

In the deep south, Alf's company monitors the temperature of a remote storage location for organically grown vegetables. The vegetables must be kept cold (above 40°F) but not frozen.

To perform the monitoring, the company, Alf's Beets, uses a simple video link to observe a standard thermometer plus an embedded-processor warning system. The embedded system consists of an eight-bit electronic sensor at the remote site that reports the temperature from the freezing point (32°F) up to 159.5°F in 0.5°F intervals.

The sensor hardware reports temperatures of 32°F and below as the value 0, and temperatures of 159.5°F and above as 255. The software measures and logs the temperature every 5 min. If the temperature drops below 40°F, a warning siren sounds.

Some typical control code, written in C, is given in Listing 1. The programmer chose `unsigned char` variables because that type's description matches the eight-bit, 0–255 value range of the temperature sensor data. This corresponds to using byte instructions if the code had been directly done in assembly.

Note that the additional software calibration correction (4°F) is built into the program so the thermometer and sensor temperatures agree. But, just as everybody is going home, the sensor temperature calibration error changes to 5°F from 4°F.

The technician ponders, "Should the calibration be fixed now or can the program be corrected tomorrow?" The technician heads home reasoning, "We might get an unnecessary false alarm if the temperature drops below 41°F, but that's not serious."

Overnight, freak weather conditions occur, plunging the temperature to 31°F. Despite this being well below the overcompensated warning level (41°F), no warning siren sounds and all the beets get beat. Alf quickly calls his insurance company, asking, "What happened to the warning signal?"

If you think it is a problem with using the 8-bit `unsigned char` variables, try casting the variables to `unsigned short int` (16 bit) or `unsigned long int` (32 bit). If you have designed a newly announced 64-bit Intel processor into your next application, then cast the `unsigned char` to `unsigned very_long int`.

If you believe the problem should be recoded with signed variables, work through the following scenario.

CHINOOKS IN CALGARY

In Calgary, Canada, Chinooks don't mean salmon or helicopters. Chinooks are warm winds that develop over the local Rocky Mountains and descend on Calgary in midwinter. Temperatures climb 30°C (80°F) or more over a period of several hours, stay high for hours or days, and then plummet back to –20°C in an equally short space of time.

Pleasant as the Chinook winds are for most Calgarians, they play havoc with the temperatures in greenhouses where plants are growing for the coming spring. Our second situation concerns a greenhouse monitoring system maintained by the Thomson sisters at Green Thoms Inc.

This temperature-control system has an eight-bit temperature sensor. Since possible air temperatures can be positive or negative, the device is designed to operate in the range –32°C to +31.75°C with a sensitivity of 0.25°C. The hardware reports temperatures above +31.75°C or below –32°C and

Listing 2—*This simple embedded-processor program reads the temperature from an eight-bit sensor and then corrects and logs temperatures in the range –32°C to +31.5°C. A warning is issued if the temperature rises above 26°C.*

```
#define CALIBRATION (4 * 4)         // Temperature correction needed
#define WARNING_LEVEL (26 * 4)      // Operate warning at 26°C
#define DEVICE_ADDRESS 0x500000     // Memory-mapped device location

// Set device pointer
char *pt = (char *) DEVICE_ADDRESS;
char value;

for ( ; ; ){
  Wait_5_minutes();
  value = *pt;                      // Read temperature
  value = value + CALIBRATION;      // Correct temperature
  Log_Temperature(value);           // Log temperature
  if (value >= WARNING_LEVEL)       // Issue warning
    Activate_Siren();
}
```

the eight-bit values 127 and –128, respectively.

Listing 2 shows the monitoring program used to ensure that the temperature does not rise above 26°C. Signed arithmetic is used because temperature values can be positive or negative. A temperature calibration correction of 4°C is built directly into the program by the software company that did the original development.

Again, just before going home, a technician notices that the sensor temperature reads 1°C higher than it should be. This implies that the alarm will sound when the true air temperature rises to 25°C rather 26°C. An earlier warning such as this is not a problem, so the technician decides to fix the code in the morning.

Overnight, the Chinook wind rises and the temperature jumps to 32°C. Despite this temperature being way above the overcompensated 25°C warning level, no warning signal sounds and the plants shrivel and die. The Thomsons call their insurance agent, "Why did this problem occur?"

Listing 3—*These embedded-controller programs are simple prototypes that can be used to demonstrate the behavior of the embedded control systems that failed.*

a)

```
#define TEST_UNSIGNED ????        //???? is part of the code
int Test_Alf(void)
{
  unsigned char value;
  int warning_signal = 0;
  value = (TEST_UNSIGNED * 2);    // Set temperature
  value = value - CORRECTION;     // Correct temperature
  if (value <= (40 - 32) * 2)     // Check temperature
    warning_signal = 1;
  return(warning_signal);
}
```

b)

```
#define TEST_SIGNED ????
int Test_Thoms(void)
{
  signed char value;
  int warning_signal = 0;
  value = (TEST_SIGNED * 4);      // Set temperature
  value = value + CALIBRATION;    // Correct temperature
  if (value >=  (26 * 4))         // Check temperature
    warning_signal = 1;
  return(warning_signal);
}
```

EXPERT WITNESS

Because of your long experience with embedded-system controllers, you're hired as an expert witness by the companies that insure Alf's Beets and Green Thoms. The insurance companies don't want to pay if there's somebody they can blame.

Possible culprits are:

- the technicians who used the WAIL (Worry About It Later) approach
- the people who built the temperature sensors
- the software company that developed the original monitoring programs
- the person who, five years ago, developed one of the software tools used during the program development. This person has since left the development company and now designs embedded-system controllers (i.e., you?).

TESTING C CODE

As an expert witness, you need to examine the code and run it through a few sample sessions. If you don't have an embedded-processor development environment, you might want to grab one of the 68k or PowerPC sampler kits from the Software Development Systems Web site.

I discussed a version of these SDS sampler kits in an earlier article ("The Evaluation Board Saga Continues," *INK* 70). These kits are more than adequate for the current task. You can purchase a full system later when you are hired to investigate larger systems.

The first thing to develop is a couple of simple test subroutines (see Listings 3a and 3b) that contain all the essential elements of the problem, setting, correcting, and then testing the temperature.

Table 1 shows how the program behaves for certain corrected temperatures for Alf's embedded processor program. If you don't believe the results, then try the code yourself with various different reported temperatures using your own compiler and debugging environment.

The problem is now obvious. The siren sounds for low temperatures 32–40°F but not for the very low temperatures 28–31.5°F.

So, if the temperature slowly drops to 31°F (41°F, 40°F, 39°F,…, 31°F), the program activates the siren at 40°F. However, in the unlikely event that the temperature drops suddenly between temperature samples (41°F, 31°F), no siren goes off.

Sensor Reading	Corrected Temperature	Action
46°F	42°F	No warning
44°F	40°F	Warning
42°F	38°F	Warning
40°F	36°F	Warning
38°F	34°F	Warning
36°F	32°F	Warning
34°F	30°F	No warning
32°F	28°F	No warning

Table 1—*In testing the Alf company code, the expert witness finds that low-temperature warning signals are only given for temperatures in the range 32–40°F. No warning signals are given for very low temperatures such as 28–31.1°F.*

A similar effect occurs with the Green Thoms' program in the unlikely event that the temperature suddenly rises into the range 32–35.75°C from a temperature below 26°C. Unfortunately, the unlikely event happened to them.

EXAMINING ASSEMBLY CODE

Knowing the circumstances under which the embedded programs fail, you now need to examine the code execution in detail. Photo 1 shows a screen capture of the code from List-

ing 3a with both C and 68k assembly code information displayed.

It is straightforward to set a breakpoint at the start of a routine and display register and memory contents while stepping through the instructions. Table 2a shows the register information corresponding to uncorrected and corrected temperature values in Alf's embedded program.

It becomes obvious that hidden behind the C code is the classic problem of representing a value that can't be encoded in the number of bits available. The same problem would occur if the code was written directly in assembler.

Using normal mathematics, we can correct a sensor reading of 32°F to be 28°F by subtracting 4°F. However, we have represented 32°F as the bit pattern 0, so that with the one bit equal to 0.5°F accuracy available on the temperature sensor, 28°F should be a bit pattern equivalent to –8. The problem is that negative numbers can't be represented as `unsigned char`.

However, during the `SUB` instruction, the processor does the best job it can. It generates and stores an incorrect value, 0xF8.

Unfortunately, the next instruction (the compare) doesn't know the stored value is the result of an incorrect operation, so it treats the value as a valid `unsigned char` that corresponds to a high positive temperature. Such high temperatures don't trigger the alarm siren.

Table 2b shows the register information corresponding to uncorrected and corrected temperature values in Green Thoms' embedded program. Here, you see that the effect of correcting a temperature of 28°C (to become 32°C) results in a temperature that can't be represented as a `signed char` value. This time, the program treats the incorrect stored value as a very low temperature, and again no alarm sounds.

Errors like these aren't caught by a compiler or assembler as the values in the registers change as the program runs. To avoid the errors, it's necessary for the programmer to add out-of-range checks after every `ADD` or `SUB`. This approach slows down code execution considerably, and a different approach should be taken when possible.

One technique is to design an algorithm where you can guarantee that no out-of-range values can occur. This option probably results in code that ensures correct results but runs slower than code using eight-bit operations.

An annoying feature of such an approach is that the out-of-range problem will occur one time in a million in real life. Therefore, all code must be slowed for the sake of safety. If speed isn't critical, then it's reasonable to adopt this approach.

Using a different processor overcomes the validity problem after arithmetic operations without a loss of speed. Some newer chips have the out-of-range check capability directly built into arithmetic instructions ("Being ASSERTive with Your Processor," *INK* 56).

There are specific `SUBU` and `SUBS` instructions for performing subtract operations on unsigned and signed numbers, respectively. Exception handlers deal with the special situations that infrequently arise.

The manufacturers of the temperature sensors designed the control logic to avoid equivalent hardware overflow conditions and associated errors. Remember the specifications:

- Alf's sensor—temperatures of 32°F and below are reported as 0, and temperatures 159.5°F and above are reported as 255
- Green Thoms' sensor—temperatures above +31.75°C or below –32°C are reported as the 8-bit values 127 and –128, respectively

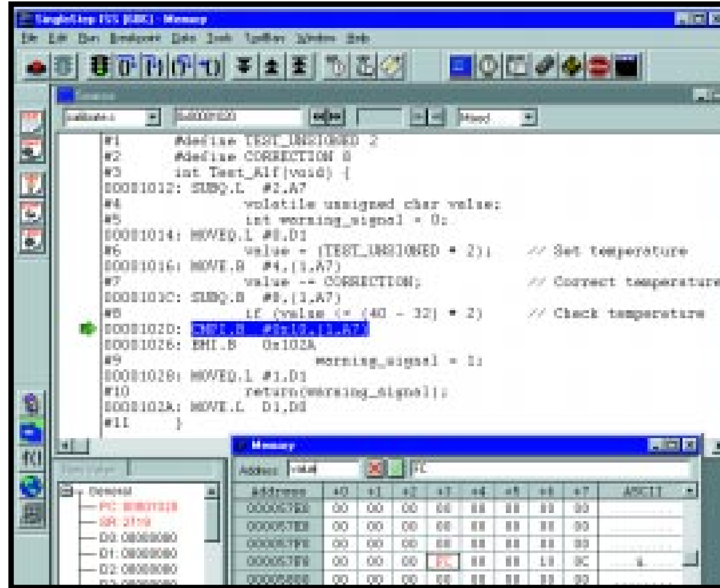


Photo 1—*This screen capture from the SDS 68k development environment shows both C code and the corresponding assembly-language code for the* `unsigned char` *operations described in Listing 3.*

No overrange problems are designed into these fictional devices. (I wonder what is designed into actual sensors?)

BASIC COMPILER ISSUES

As a defense witness for the insurance company, you come into court prepared to explain why the programs failed to generate the expected warning signals. You think the sensor manufacturer should be let off the hook.

Since the problem arose from the temperature correction already in the code, then the technicians may be fired for procrastination but they're not culpable for damages. On the other hand, the people who added the correction to the code shouldn't be sleeping so easily.

The prosecuting attorney stands up (a computer-engineering major who switched over to law to use domain knowledge and make a fortune when year-2000 lawsuits start appearing).

Counsel begins, "How do you know that the compiler you're using to demonstrate the problems generates the same code as the compiler used in developing the original code?"

Good point. Compilers must, or rather should, follow the same basic rules for the C language. However, that doesn't mean that the actual lines of assembly code they generate are the

same—just the final results from running the code.

For convenience, let's assume you can reassure counsel that you have example listings and other documents to prove that the compilers generate identical code.

The prosecutor then asks, "What optimizations did you activate in the C compiler? Were they the same as for the original code?"

This is a key issue when using C to develop an embedded-processor system. Unlike normal programs that access RAM locations, embedded programs access memory-mapped peripherals, whose values change from an external influence rather than through a direct programmed action.

Under normal situations, the code:

```
int temperature, number, loop;
for (loop = 0; loop < 9; loop++)
  number = temperature;
```

can be optimized to:

```
int temperature, number;
number = temperature;
```

The contents of `temperature` never change inside the loop, so the (constant) operation can be brought outside the loop. The loop is now dead code and can be optimized away.

If this type of loop optimization was happening in the monitoring system, then it's not surprising that things went wrong. The C code might be written to continually access the temperature sensor, but after optimization the sensor is only accessed once and all temperature changes are ignored.

There are various ways to solve this problem. A programmer can inform the compiler that external influences may change a variable value so that operations with this variable should be optimized carefully. The keyword `volatile` should be used in this situation:

```
volatile int temperature;
int number, loop;
for (loop = 0; loop < 9; loop++)
  number = temperature;
```

Or, the programmer can simply turn off the optimizations.

	Sensor Temperature	Sensor Reading	Sensor Reading After Correction	Corrected Temperature	Action
a)	46°F	0x1C	0x14	42°F	No warning
	44°F	0x18	0x10	40°F	Warning
	42°F	0x14	0x0C	38°F	Warning
	40°F	0x10	0x08	36°F	Warning
	38°F	0x0C	0x04	34°F	Warning
	36°F	0x08	0x00	32°F	Warning
	34°F	0x04	0xFC	158°F	No warning
	32°F	0x00	0xF8	156°F	No warning
b)	30°C	0x78	0x88	–30°C	No warning
	28°C	0x70	0x80	–32°C	No warning
	26°C	0x68	0x78	30°C	Warning
	24°C	0x60	0x70	28°C	Warning
	22°C	0x58	0x68	26°C	Warning
	20°C	0x50	0x60	24°C	No warning

Table 2a—*In the Alf company code, some low temperatures are incorrectly represented as high temperatures so that no warning signal results.* ***b***—*In the Thom company code, some high temperatures are incorrectly represented as low temperatures so that no warning signal is given.*

In either of these cases, the safest bet is to physically examine the assembly code the compiler generates (see Photo 1) and check whether the access to hardware is being performed correctly. The keyword `volatile` is already in Photo 1. Without it, the SDS compiler would optimize this simple test code down to a single instruction.

SDS suggests a third approach—access all hardware directly using assembly-language routines. If you use a standard convention for register and stack use, then it's straightforward to link assembly-code subroutines to the rest of your C code.

This combination of interfacing C with assembly code, and using assembly code to access hardware, is a good approach. If a picture is worth a thousand words, then one line of C is worth a thousand lines of assembly code, besides being much faster and more accurate to develop.

TECHNICAL COMPILER ISSUES

Prosecuting counsel now gets technical, "I notice in Photo 1 that the assembly code involves only eight-bit operations. Does this go against the C standards for type conversion?"

Although the judge has probably lost track of the argument, you haven't. This code explains the concepts referred to:

```
unsigned char value;
char temperature;
value = value - 8;
temperature = temperature + 16;
```

which the 68k compiler translates as:

```
SUB.B #8, value
ADD.B #16, temperature
```

Other compilers would do something equivalent for any processor.

The C standard [1] states that the default type for any value is `int` (which is 32 bits with the SDS compiler unless you specify otherwise). Thus, the value `8` in the subtraction is actually `8L`.

By default then, you are subtracting a 32-bit (`signed`) `int` (the number `8L`) from an eight-bit `unsigned char` (the variable value). The ISO C standards imply that the eight-bit `unsigned char` must be properly converted to a (`signed`) `int` before the subtraction is performed, and the result converted back to an `unsigned char`.

Counsel is hinting that this conversion to 32-bit `int` values doesn't appear to happen during the operation `SUB.B #8, value` and the `ADD.B #16, temperature` instructions. Should the compiler have generated the code in Listing 4, which does explicit conversions?

If you have assembly-language experience, you've probably never thought of byte (`.B`) operations like this before. You start getting worried.

Fortunately, your ignorance on bytes isn't publicly exposed.

PIVOTAL ARGUMENTS

Table 3 shows that whether value is the `unsigned char` number 7 or

Subtracting using 8-bit operations only		
Initial internal representation of value	0x??????07	Unsigned decimal 7 (8 bits)
Final internal representation of value	0x??????FF	Unsigned decimal 255 (8 bits)
Initial internal representation of value	0x??????07	Signed decimal –7 (8-bits)
Final internal representation of value	0x??????FF	Signed decimal –1 (8-bits)
Subtraction when directly implementing extensions to int (32-bits) types		
Assuming value was an unsigned char variable		
Initial internal representation of value	0x??????07	Unsigned decimal 7 (8 bits)
D0 – (signed int) value	0x00000007	(32 bits)
D0 – after 32-bit subtraction	0xFFFFFFFF	(32 bits)
Final internal representation of value	0x??????FF	Unsigned decimal 255 (8 bits)
Assuming value was a signed char variable		
Initial internal representation of value	0x??????07	Signed decimal 7 (8 bits)
D0 – (signed int) value	0x00000007	(32 bits)
D0 – after 32-bit subtraction	0xFFFFFFFF	(32 bits)
Final internal representation of value	0x??????FF	Signed decimal –1 (8 bits)

Table 3—*The same result occurs whether the operation* `value = value - 8` *is performed directly using eight bits or explicitly extending* `signed char` *and* `unsigned char` *type to* `int` *types. Similarly, the same result occurs if the operation* `temperature = temperature + 16` *is performed directly using eight bits or explicitly extending* `signed char` *and* `unsigned char` *type to* `int` *types.*

the `signed char` number 7, then the instruction `SUB.B #8, value` produces the same eight-bit final result (0xFF) as if the compiler had developed an explicit series of `char` to `int` and `int` to `char` operations (see Listing 4). This is true despite the fact the final `unsigned char` value is an invalid representation of the result.

Similarly, it can be shown that whether temperature is the `unsigned char` number 120 or the `signed char` number 120, then the operation `ADD.B #16, temperature` produces the same 8-bit final (0x88) as if the compiler had explicitly done conversions. Again, the operations are equivalent despite the `signed char` number representation overflow.

However, the byte operation is equivalent to a series of `int` conversions only if you consider the final result. If you also take into account the internal operation of a 68k processor, then the byte `SUB` or `ADD` operations set the `C`, `V`, `N`, and `Z` flags, whereas the final `MOVE` instruction from the series of conversions only sets `N` and `Z` flags. Other processors may effect their flags in different ways.

FINAL SESSION

After hearing your explanation that a standard `SUB.B` operation provides an implicit version of the required type conversions, the judge asks how you would have written the temperature controller code to avoid the no-warning problem.

With relish, you produce the already developed solution (i.e., Listing 5) from your briefcase. The solution is to develop code that uses an intermediate `short int` variable (signed 16 bit) to ensure that out-of-range errors don't occur at any point in the algorithm.

The corrected solution adds just one line to the original code. On the

Listing 4—*These code segments demonstrate how to explicitly extend 8-bit* `unsigned` *and* `signed` *byte values to and from (*`signed`*)* `short int` *values while performing mathematical operations.*

```
; unsigned char value - code for value = value - 8
  MOVE.L #0, D0
  MOVE.B value, D0           // D0 = (int) value
  SUB.L #8, D0               // D0 = D0 - 8
  MOVE.B D0, value           // value = (unsigned char) D0

; signed char value - code for reading = reading + 16
  MOVE.B temperature, D0     // D0 = (int) temperature
  EXTB.L D0
  ADD.L #16, D0              // D0 = D0 + 16
  MOVE.B D0, temperature     // temperature = (signed char) D0
```

Listing 5—Making use of an intermediate `short int` variable `temperature` and changing the function `Log_Temperature()` is all you need to make the embedded-processor code safe for Alf's control programs. Equivalent changes correct the Green Thoms' code.

```
// Set device pointer
unsigned char *pt = (unsigned char *) DEVICE_ADDRESS;
unsigned value;
short int temperature;
for ( ; ; ){
  Wait_5_minutes();
  value = *pt;
  temperature = (short int) value;  // New-Cast to short int
  temperature = temperature - CORRECTION;
  Log_Temperature(temperature);
  if (temperature <= WARNING_LEVEL)
    Activate_Siren();
}
```

68k processor, the speed penalty is small because the data bus is 16 bits wide, so 8-bit (`.B`) and 16-bit (`.W`) operations execute in the same amount of time.

You are dismissed as a witness and go home to enjoy your fee. The case has far to go before blame can be duly placed.

MY VERDICT

I've illustrated the damage done by a simple hidden calibration fault in a temperature sensor used to monitor the health of a pile of vegetables. If you want to consider a more expensive problem, monitor the health of a $260-million revolving stator at a generating station.

On a more personal note, I imagine the calibration correction in the fetal-monitor checking the heart rate while my daughter was being born. I'm glad that designer got it right.

But, what about the child born January 1, 2000? Will that child's monitor suddenly decide it has not been serviced for 100 years and, for safety reasons, switch itself offline?

The calibration bugs I've mentioned arise from an incomplete understanding of the background behind the embedded technology we're using. It's better to avoid such defects now than to fix them later.

This paraphrase from *The Economist* is applicable to many bugs other than just the millennium one: This bug is just that, a bug. In fact, it is a fairly straightforward bug to fix. The only problem is that it is going to be time consuming and expensive to find. Many of the problems it causes will go unnoticed.

Make sure that you and your firm are part of the solution—not the embedded-controller problem! ■

Many thanks to Dr. Steven Norman, University of Calgary, who pointed out the explicit and implicit 32-bit conversions using byte operations. Thanks also to Mukesh Chitroda, a student in my microprocessor class who delighted in the chance to correct my work rather than the other way around.

Mike Smith is a professor in the computer engineering department at the University of Calgary. Mike is interested in applications of embedded processor systems and is currently working with students on a voice-recognition system for a computerized sailboat built by the Alberta Disabled Sailing Assn. His nonstandard approach to teaching assembly-language programming using C and C++ is viewable at www.enel.ucalgary.ca/People/Smith/index.htm. You may reach him at smith@enel.ucalgary.ca.

REFERENCE

[1] S.P. Harbison and G. Steel, Jr., *C: A Reference Manual*, 4th ed., Prentice-Hall, Englewood Cliffs, NJ, 1995.

SOURCE

68k or PowerPC sampler kits
Software Development Systems, Inc.
(800) 448-7733
(630) 368-0400
Fax: (630) 990-4641
www.sdsi.com

FEATURE ARTICLE

Rick May

A PIC-Based AC Power Meter

Questioning your power bill? Rick shows you how to build a tool to make sure your power bill stays right on target—a portable AC power meter that displays the power delivered to and consumed by your house.

When I watch the old mechanical wattmeter on the house spin, I marvel at just how low tech this device is. No fancy displays, no RF transponders sending telemetry to roving meter-reader trucks (at least not in my neighborhood).

I always wondered why I couldn't go out and buy a small hand-held instrument that I could use to figure out just what was causing that old-fashioned meter outside to spin so fast.

A few years ago, I came across a novel circuit design by Stephen Woodward [1]. He used a quad optoisolator (conventionally a nonlinear device) to generate an analog voltage proportional to the power consumed by a load. It used optos for safety, and, well, it was just neat.

So, off I set to marry some type of micro to that analog front end. I wanted to build a hand-held, portable AC power meter that could display the power delivered to a load. The result: see Photo 1.

By tossing in a little math and some numerical integration, I could also display the energy consumed by a load. For a little added challenge, I used a PIC microcontroller that has no multiply or divide instructions.

SYSTEM DESIGN

I want my power meter to measure AC power, instantaneous and average, 0–1200 W, as well as measure AC energy consumption in kilowatt-hours (or watt-hours). Additionally, I want it to provide digital readout of power or energy, and it should be easy to hook up using standard power receptacles and plugs.

This device is to be packaged as a hand-held instrument, and my budget dictates some ultra-low-cost parts.

I designed the power meter around the Woodward power-measurement circuit. The power source is a 9-V battery, instead of stealing power from the AC line. This setup ensures maximum isolation from the AC line and increases safety.

I chose a nonmultiplexed LCD because of its low power consumption. I use a simple four-digit, seven-segment type rather than an alphanumeric LCD module because I only need to display numeric data. It also costs less.

The user operates the meter using two momentary switches. The mode switch causes the display to cycle though the different operational modes. Regardless of what operational mode the power meter is in, energy-consump-

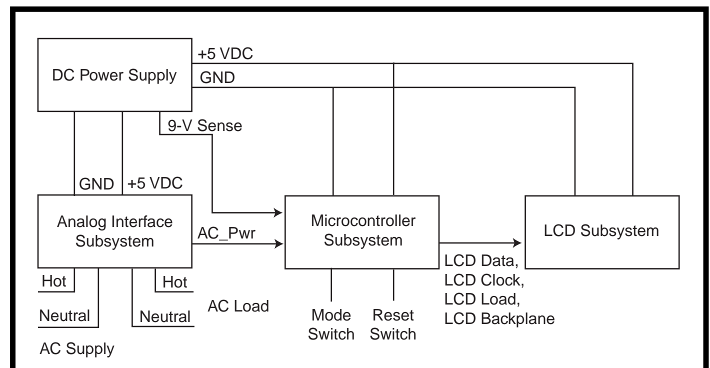


Figure 1—*The power meter is broken down into four subsystems. The user interface is accomplished with just two switches and an LCD.*

Photo 1—*Here's my dream come true—a hand-held portable AC power meter.*

tion accumulation still occurs. The reset switch resets the energy-consumption accumulator in any operation mode.

This power meter has four operational modes. The power mode displays the power consumed in watts. By accumulating power over time, the energy mode displays watt-hours or kilowatt-hours. The average power mode displays energy consumed over time in watts. The remaining mode displays the time elapsed since reset in hours, minutes, and seconds.

Because the power meter toggles through these four modes, the user needs to know the current operational mode. Normally, this would be done with annunciator segments in the LCD. However, I want to use low-cost stock parts, so a custom LCD with annunciator segments isn't an option.

The LCD I chose is a four-digit display with three decimal points, a colon, and an arrow. Because I have no real use for the arrow, I decided to use it as the mode indicator.

The colon is used in the time mode, so I only need to discriminate among three modes—power, energy, and average power. I decided that no arrow indicates power mode, a slow-flash arrow means energy mode (1-Hz rate), and a fast-flash arrow is used for the average-power mode (2-Hz rate).

HARDWARE DESIGN

Figure 1 shows the four subsystems of the power meter. The power-supply subsystem supplies DC power to the other subsystems. The analog interface contains Woodward's circuit [1].

The microcontroller and A/D subsystem acquire data and compute power readings that are then displayed by the LCD subsystem. Figure 2 shows how the subsystems link together.

A 9-V battery connects to J6 that feeds a 78L05 regulator. The 5.1-kΩ 1% resistors in a voltage divider network provide the 2.5-V bias voltage for the analog section. They also provide the 9-V sense voltage to the microcontroller for low-battery detection.

A resistor divider network is selected over a 2.5-V reference diode because of cost to provide the 2.5-VDC reference used in the analog section. I chose 5.1-kΩ resistors since I'm already using this value in the analog interface.

Woodward's original circuit used ±15-VDC power supplies and an OP27 precision op-amp. My goal was to modify his circuit using a +5-VDC single supply and a common LM358 op-amp.

The redesign for single-supply operation is straightforward: swap 2.5-V DC bias for ground, ground for –15 V, and +5 V for +15 V. Woodward's circuit indicates power delivered to and from the load, with power delivered to the load being below the 2.5-V bias.

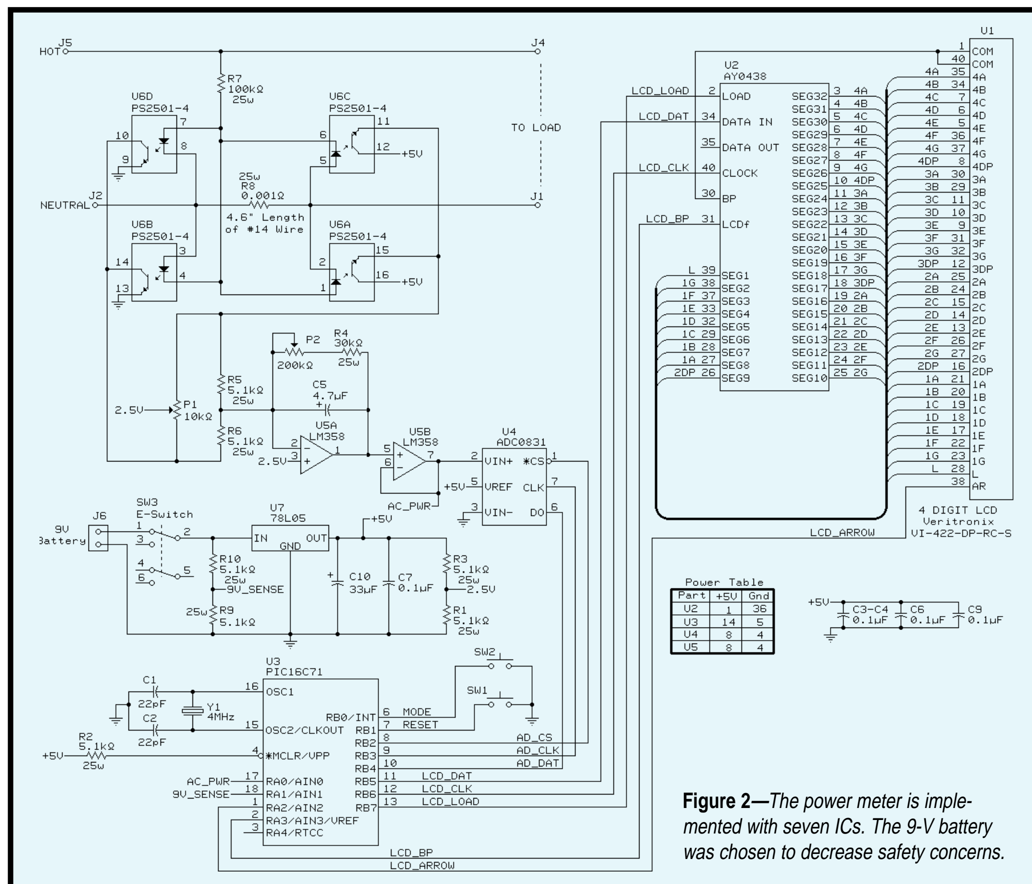
Power readings range between ground and +2.5 V, effectively reducing the ADC resolution to seven bits. Therefore, 128 discrete output values are possible between no load and full scale.

A pot, P1, calibrates the full-scale value to approximately 1200 W nominal. By adjusting the pot to give a full-scale reading of 1280 W, the actual power (in watts) is obtained by:

$$\text{power} = (0x80 - \text{ADvalue}) \times 0x0A$$

So, if *ADvalue* equals 0x76, the power is 100 W.

I initially chose the PIC16C71 because of its low cost and onboard



Part	+5V	Gnd
U2	1	36
U3	14	5
U4	8	4
U5	8	4

Figure 2—*The power meter is implemented with seven ICs. The 9-V battery was chosen to decrease safety concerns.*

a)

	7	6	5	4	3	2	1	0
PORT_A	N/A	N/A	N/A	X	ay_bp	ay_arr_out	9V_sense	ac_pwr

ac_pwr	Analog input	Analog AC power input (PIC16C71 only)
9V_sense	Analog input	Analog 9-V battery sense (PIC16C71 only)
ay_arr_out		Arrow segment pin of LCD
ay_bp		LCD backplane input pin

b)

	7	6	5	4	3	2	1	0
PORT_B	ay_ld	ay_clk	ay_dat	ad_dat	ad_clk	ad_cs	res_sw	mode_sw

mode_sw	Active low	Indicates mode switch depressed
res_sw	Active low	Indicates reset (or zero) switch depressed
ad_cs	Active low	Chip select to ADC0831
ad_clk	Falling edge	Data valid out of ADC on falling edge
ad_dat		Data output of ADC0831
ay_dat		Data input to AY0438 LCD driver chip
ay_clk	Falling edge	Clock input to AY0438, data is clocked into AY0438 on falling edge
ay_ld	Rising edge	Data is transferred from AY0438 shift register to output latches

Figure 3a—*Here are the port A assignments. With just 13 I/O pins available, allocation is critical.* ***b***—*With the port B pin assignments, just three pins are needed for the A/D conversion, but five are required for the LCD controller.*

four-channel eight-bit ADC. However, you can save even more by replacing the '16C71 ($12.30) with a separate ADC, National's ADC0831 ($3.29), in combination with the ADC-less PIC16C61 ($6.15).You save $2.86, but you need space for one more 8-pin DIP.

The National ADC0831 is a low-cost 8-bit ADC that has a simple three-wire serial interface (chip select, clock, and data) and a 32-µs conversion time.

Note there are no pull-up resistors on the two momentary switches. Pullups are provided internally by the PIC16C61. Again, you see the 5.1-kΩ 1% resistor used as the MCLR pullup, which means the MCLR pin can be shorted to ground without shorting out the +5-V supply.

This component could be replaced with a 0-Ω jumper for production. Currently, a crystal is used as the microcontroller clock, but you could replace it with a ceramic resonator for more cost savings.

The display subsystem uses a nonmultiplexed Varitronix LCD, with a Microchip AY0438 LCD driver. The AY0438 operates up to 32 segments of an LCD, providing a simple three-wire serial interface (data, clock, and load), and it's capable of generating the AC waveforms required to illuminate LCD segments.

For a LCD segment to be on, there must be voltage differential between the segment pin and the backplane. However, this voltage cannot be static (non-time-varying) or the display gets permanently damaged. To drive the LCD correctly, a low-frequency (100 Hz) square wave is applied to the backplane pin.

For a segment to be on, a segment pin must have the inverted backplane signal applied. For a segment to be off, the segment pin must have the in-phase backplane signal applied.

The AY0438 generates this backplane waveform and the correct segment waveforms without processor involvement. Just hang a capacitor on the LCDf pin to control the onboard oscillator.

Initially, I used the onboard oscillator of the AY0438 to generate the AC waveforms needed and to get the display up and running. However, the AY0438 can only drive 32 segments.

I needed 33 segments to use the arrow segment and all four digits, three decimals, and the colon. With a couple of unused pins on the PIC, I figured it couldn't be that hard to drive the LCD directly.

And, driving the LCD is easy. Just remove the cap from the LCDf pin, then drive this pin (backplane) and the thirty-third segment directly from the PIC. Remember that you can't drive the LCD segment pin alone because the AY0438 oscillator (backplane) would be asynchronous with respect to the PIC.

The software must toggle the backplane and the thirty-third segment every 10 ms. Note that the AY0438 still manages the other 32 segments

and provides a serial interface. The AY0438 just gets its backplane frequency reference from the PIC. This is a case where a watchdog timer should be used because display damage can result if the backplane doesn't toggle at least every 10–100 ms.

Register definitions of external interfaces to the PIC are shown in Figure 3. The PIC16C61 is an 18-pin DIP and only has 13 I/O pins. Obviously, when you're using an I/O-limited micro like the PIC, it's important to choose peripherals with low pin-count interfaces.

Five pins are used for the LCD subsystem. It could have been three if I didn't need the thirty-third segment. Three pins are used for the ADC and two for the mode switches.

SOFTWARE DESIGN

The PIC's internal timer generates a 10-ms timer interrupt for periodic software functions. This works well since I need to service the LCD backplane in the 10–100-Hz range, and it also lets me sample power at a 100-Hz rate.

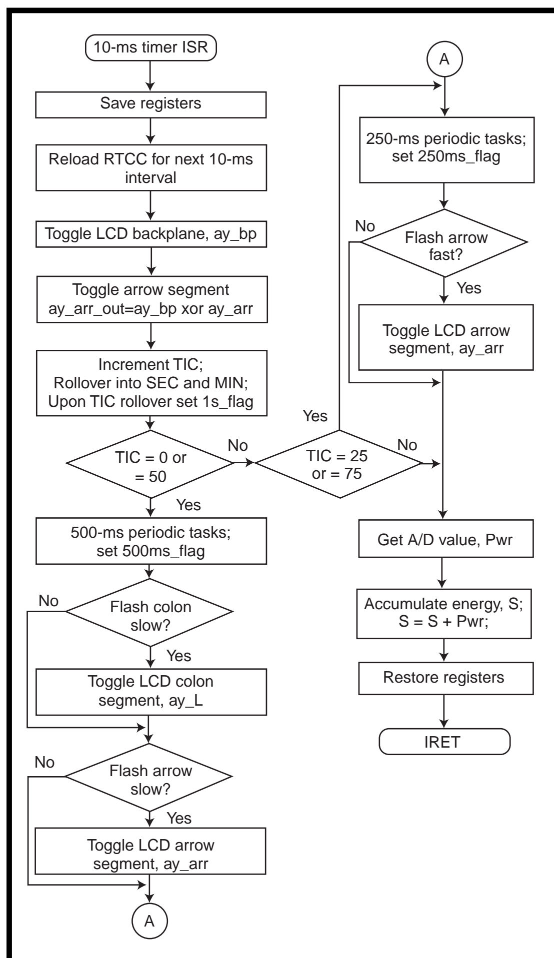


Figure 4—*For every 10-ms period, energy accumulates in S until the* `1s_flag` *is set.*

Figure 4 shows a flowchart of the 10-ms timer ISR. Every 10 ms, the LCD backplane and arrow segment must be toggled, the tic (0.01 s) counter incremented, and power value accumulated into *S*.

Every 250 ms, the fast flash of the arrow segment is done if needed. Every 500 ms, the slow flash of the arrow segment is done if necessary, and the power value computed for display.

Figure 5 illustrates the main-loop processing. Both switches are scanned every time through the main loop, debounced through a 50-ms delay.

Every second, the larger 24-bit energy accumulator, *E*, is updated from the local 16-bit accumulator, *S*. The mode state variable directs execution to the appropriate processing for the four operational modes.

MEASUREMENT CONSIDERATIONS

The raw value from the ADC is read at a 100-Hz rate or every 10 ms and is in the 0–127 range. This value indicates a 10-W increment of power (so, 12 means 120 W). Let's refer to this raw A/D value as having the unit of a ten-watt (i.e., 10 W).

To display a power reading, the main loop reads the current value of power that the ISR writes to, multiplies this value by 10, converts this result to a five-digit BCD word, and then displays the four least significant digits of the result. The multiply requires 16-bit math because 10×127 equals 1270, which doesn't fit into 8 bits.

I found the 16-bit math and BCD conversion routines on the Microchip BBS. The routines are fairly compact in size and require approximately 300 cycles (300 µs at 4 MHz) for an unsigned multiply or divide.

Since the 16-bit values need to be converted to BCD prior to display, and the BCD routines return a five-digit result, a divide by 10 is done just by displaying the upper four digits instead of the lower four digits. This fact is taken into account when designing scaling algorithms.

For energy consumption, a 24-bit accumulator is required to capture a reasonably large amount of energy consumption, given the 100-Hz accumulation rate. To keep the non-reentrant math routines out of the timer ISR, the timer ISR accumulates the power value (0–127) into an intermediate 16-bit accumulator, *S*. The main-loop routine adds *S* to the master 24-bit accumulator, *E*, when the `1_sec_elapsed` flag is set.

The *S* value is in units of tenwatt-seconds. To accumulate a wide range of energy consumption, the 24-bit *E* accumulator is in units of milliwatt-hours. This allows a maximum value of $2^{24} - 1 = 16,777,215$ mWh or 16.77 kWh.

The conversion from tenwatt-seconds to milliwatt hours is:

$$\text{milliwatt–hour} = \text{tenwatt–second} \times \frac{1\ \text{h}}{3600\ \text{s}} \times \frac{10\ \text{W}}{1\ \text{tenwatt}} \times \frac{1000\ \text{mW}}{1\ \text{W}} = \text{tenwatt–second} \times \frac{100}{36}$$

However, since the *S* accumulator represents 100 samples over a 1-s period, we must divide *S* by 100 before this conversion. So, the *S*-to-*E* accumulation calculation (performed only once per second) is:

$$E = E + \frac{S}{36}$$

in milliwatt-hours. Note that this conveniently reduces a multiply and divide operation to a single divide.

To filter transients in power-measurement mode, a 500-ms moving average is implemented. The *S* accumulator is restarted every second as part of the energy-consumption function. At 500 ms after restart, a power value is calculated by:

$$\text{power (watts)} = \frac{S}{50} \times 10 = \frac{S}{5}$$

At 1 s after restart, a power value is calculated by:

$$\text{power (watts)} = \frac{S}{100} \times 10 = \frac{S}{10}$$

To ensure that no-load situations read zero and that noise does not cause false readings, hysteresis is added to cause readings less than 2 (20-W reading) to be treated as zero. This also helps in setting the zero trim pot.

DISPLAY CONCERNS

Now that I've discussed how the *E* accumulator is maintained, I want to turn to how the energy accumulation is displayed.

If watt-hour readings are the finest resolution displayed, then the user might have to wait 2 min. to see 0.001 kWh, depending on the load. To solve this problem, I implemented autoranging using three display modes.

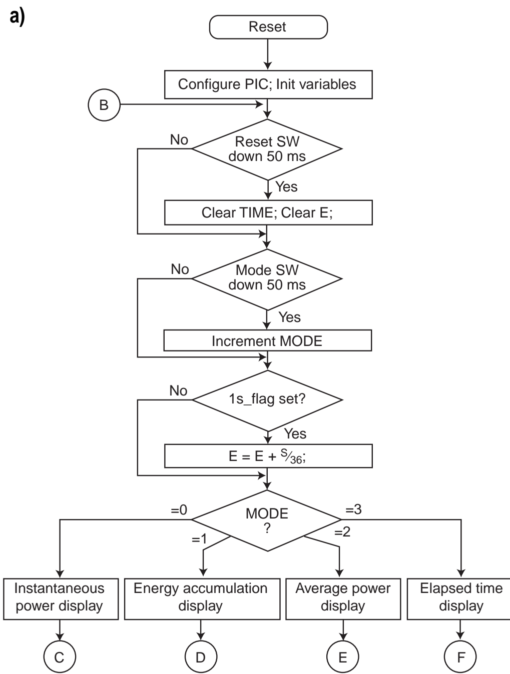
Display mode d1 displays readings in the range of 0.000–9.999 Wh. Once 10 Wh are accumulated, display mode automatically switches to d2 mode.

The d2 mode displays readings in the range of 0.010–9.999 kWh. Once 10 kWh are accumulated, the display mode automatically switches to display mode d3, which displays readings in the range 10.00–16.77 kWh.

Display mode d1 is implemented by converting the low-order 16 bits of *E* (milliwatt-hours) to BCD, then displaying the four least significant digits with the decimal point three places to the left (i.e., 0.000). The move of the decimal point effects a multiply by 1000, causing watt-hours to be displayed.

Display mode d2 requires display of kilowatt-hours, and has a least significant digit of watt-hours. Divide *E* (in milliwatt-hours) by 1000 to get the result in watt-hours. This is implemented by using the upper 16 bits of *E*, effecting a divide by 256 operation on *E*.

Recall that canned math routines support 16-bit math, not 24-bit math. So, by taking the upper 16 bits of *E* and performing a divide by 4, a divide by 1024 is implemented, which approxi-



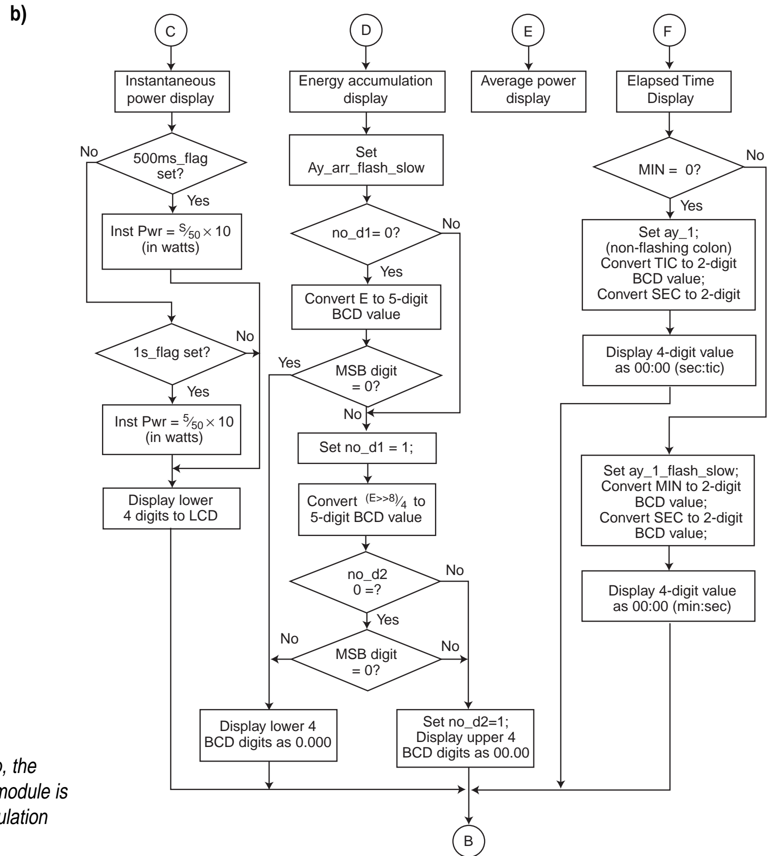


Figure 5a—*As you can see in the power meter's mail executive loop, the energy accumulator E is filled from S every second.* **b**—*Here, each module is detailed. Note how autoranging is implemented in the energy accumulation mode.*

mates the correct divide by 1000. This result is converted to BCD, and then the four least significant digits are displayed with the decimal point set three places to the left.

Display mode d3 also requires display in kilowatt-hours. The only difference between the d2 and d3 modes is that d3 needs an additional divide by 10 to provide a reading in the range (10.00–16.77). By taking advantage of the fact that the BCD conversion routine returns a five-digit result, a divide by 10 is done just by displaying the four most significant digits and placing the decimal point two places to the left.

The LCD services are designed such that the LCD routines expect data where the BCD math routine deposits its result. Although I don't elaborate on these routines in detail here, I mention them because they cooperate with the PIC math routines well.

MY TOP PIC

The PIC software occupies 722 of 1024 words of program memory. I haven't yet implemented average power measurement or low-battery detection. And, I did the initial calibration using incandescent light bulbs as loads.

The results appear accurate within 10 W, as Woodward's article indicates. Calibration using the two trim pots was fairly easy. For easier calibration, a multi-turn pot could replace the single-turn trim pot.

Some possible enhancements include use of a higher resolution ADC for more accuracy, removal of pots altogether, larger energy consumption accumulator, and autocalibration.

You just saw how a simple PIC with Microchip's math routines can be made to do significant computation such as numerical integration. You can use this project as starting point for any instrumentation-type project.

And maybe now I can figure out just what's making my wattmeter spin so fast. ■

Rick May is a principal design engineer at Raytheon Systems. He currently designs embedded software for a Navy communications system. You may reach him at rmay@televault.com.

SOFTWARE

Source code for this article can be downloaded from the Circuit Cellar Web site.

REFERENCE

[1] W.S. Woodward, "Optical isolator computes watts," *Electronic Design*, 102–103, October 14, 1994.

SOURCES

PIC16C71, PIC16C61, AY0438
Microchip
(602) 786-7668
Fax: (602) 786-7277
www.microchip.com

ADC0831
National Semiconductor Corp.
(800) 272-9959
(408) 721-5000
Fax: (408) 721-2233
www.national.com

LCD
Varitronix
(213) 738-8700
Fax: (213) 738-5340
www.varitronix.com

Designing for Smart Cards

FEATURE ARTICLE

Carol Hovenga Fancher

Part 1: What's a Smart Card All About?

They look like a credit card, but the microcontroller in them provides computational ability and stores information. Carol covers all the smart-card basics you need to know before you implement them in a design.

A smart card doesn't look so different from a credit card. But, it has an embedded controller that provides computational capability and protected storage.

A smart card's most important feature is the higher level of security it offers compared to other technologies like magnetic-stripe or memory cards. Smart cards are good for applications needing a portable token and the ability to manipulate the data they carry.

Smart cards are also referred to as an integrated circuit card (ICC), and can interface with a point-of-sale terminal, ATM, or card reader integrated into a phone, computer, vending machine, or other appliance. As Figure 1a shows, the semiconductor devices on a smart card attach to a module embedded in the top left corner of the card, which provides contacts to the outside world.

Although most smart cards require physical contact between the card and the pins in a reader, a growing number of applications use contactless cards. These cards communicate and are powered by radio signals or inductive or capacitive coupling (see Figure 1b).

Contactless smart cards are used in situations requiring quick transactions (e.g., mass-transit turnstiles). They can be more physically robust than contact cards because there's no wear and tear on the contacts and the readers aren't as open to wear or vandalism. Efforts are underway to standardize hybrid cards for contact and contactless systems.

The international standards for smart cards have been developing since the late 1970s. ISO 7816, the basis of most smart card-related standards, defines the mechanical, physical, electrical, and handshake interface between the card and reader without restricting the silicon in the card or the application for the card. More recent standards address new technologies such as contactless smart cards or application areas like financial cards, Internet payments, airline ticketing, and so on (see Table 1).

COSTS AND BENEFITS

Current smart cards, made by GemPlus, Schlumberger, and Bull CP8, among others, range in price from less than $1 to about $20. This cost includes the silicon, OS, module (the chip package providing the connections to the outside world), and plastic card.

In addition to the card itself, the software and networks previously designed to handle cash, credit, or checks have to be modified. Let's look at the benefits of implementing a financial smart card.

A stored-value card is attractive because it reduces the amount of change the shopper carries and can be used in small-value transactions where credit cards or checks are less desirable. Retailers prefer stored value because it increases small cash transactions, which financial institutions currently avoid because the overhead on credit cards or checks are too high for profit.

The cards also reduce the hidden cost of handling, storing, and safeguarding cash (estimated as ~4% of the value of all transactions).

OVERALL SYSTEM SECURITY

The security of any application depends not just on the smart card chip and its security features but on the software structures implemented on-chip and even more broadly on the integrity of the overall system.

To design for security, first define the entire system. Consider the operating environment, including any

expected, imagined, or feasible security attacks. Be paranoid. If the system involves any monetary value or secret, proprietary, or private information, there will be active attempts on the system.

Define the personality of the attacker (university student hacking for the challenge, international cartel searching for industry secrets), the attacker's resources (home workshop, university lab, or the resources of an entire government), and the value of the information to the attacker in time and money.

No security strategy is absolute. Given enough time, resources, intelligence, and luck, it's possible to circumvent any security.

Most systems impose many barriers so that defeating one or a few security features does not compromise the entire system and so that the time and resources needed to break into the system exceeds its value to the attacker. But of course, system developers need to design a reasonable and practical system with a cost commensurate with the value of the protected information.

An attacker will search out the weakest link in the security chain. So, evaluate all aspects of the system:

- is system knowledge controlled or segregated so that no one person or group knows all details?
- is the exchange or storage of information protected?
- do the protected secrets affect the entire system or a single user?
- is the system prepared to not only prevent a security break but recognize if one has taken place and have the means to recover?
- can you update the system against new attack scenarios, so the system won't become obsolete over time?

It's good to evaluate system performance using various security criteria—those of a recognized body (e.g., ITSEC) or industry (e.g., SET), or those defined only for the specific application.

Also consider the exportability of the system if the application is international or to be exported. Most governments closely control encryption or decryption techniques.

Once you identify the overall system security needs and vulnerabilities, you can use the smart card as a tool to strengthen security.

THE MICROCONTROLLER

As illustrated in Figure 2, today's smart card controller typically includes an 8-bit CPU, 128–780 bytes of RAM, 4–20 KB of ROM, 1–16 KB of EEPROM on a single die, and (optionally) an on-chip hardware encryption module.

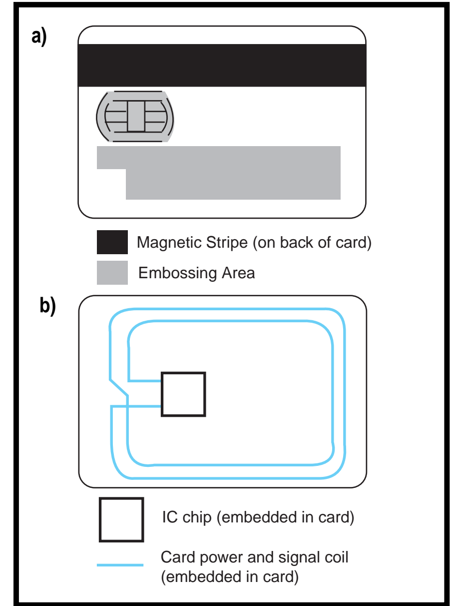


Figure 1a—*This is the plastic form factor and module for a contact smart card as defined by ISO 7816.* ***b***—*In a contactless smart card, the antenna is generally located around the perimeter of the card.*

The EEPROM is ideal for this application since the stored data usually changes over the card's lifetime or is unique to the card, such as a card identification number, a PIN (personal identification number), authorization levels, cash balances, credit limits, and so on.

This year, improvements to the controller include advanced RISC cores and increases in memory sizes to 32 KB of ROM or EEPROM.

Standard	Title/Description
ISO Standards for Identification Cards	
ISO 7810	Identification cards, Physical characteristics
ISO 7811	Identification cards, Recording techniques (6 parts)
ISO 7812	Identification cards, Identification of issuer (2 parts)
ISO 7813	Identification cards, Financial cards
ISO 10373	Identification cards, Test methods
ISO 7816	Identification cards, Integrated circuit(s) cards with contacts (6 parts)
ISO 10536	Identification cards, Contactless (Close Coupling) Integrated Circuit(s) Cards (CICC) (4 parts)
ISO 14443	Identification cards, Contactless (Remote Coupling) Integrated Circuit(s) Cards (4 parts)
ISO 15693	Identification cards, Contactless (Vicinity Card) Integrated Circuit(s) Cards (4 parts)
General ISO Security Standards	
ISO 9796	Information technology, Security techniques, Digital signature giving message recovery
Industry-Specific Standards (Financial, Telecommunications, Airline Industries)	
ISO 9992	Financial transaction cards, messages between the integrated circuit card and the card accepting device (2 parts)
ISO 10202	Security architecture of financial transaction systems using IC cards. (8 parts)
EMV	Integrated Circuit Card Specifications for Payment Systems developed by Europay International S.A., MasterCard International Inc., and Visa International Service Association (3 parts)
ETSI GSM 11.11	European Digital Cellular Telecommunications System (Phase 2): Specification of the Subscriber Identity Module—Mobile Equipment (SIM-ME) Interface
ETSI GSM 11.14	European Digital Cellular Telecommunications System (Phase 2+): Specification of the Subscriber Identity Module—Mobile Equipment (SIM-ME) Interface for SIM Application Toolkit
ANSI T1P1	U.S. Telecom Standard
IATA JPSC 791	International Airline Transportation Association (IATA) Joint Passenger Service Committee (JPSC) Smartcard Specification

Table 1—*Various organizations are involved in developing standards relating to smartcards. The Smart Card Forum has prepared an overview and description of pertinent standards, "Standards and Specifications of Smart Cards: An Overview."*

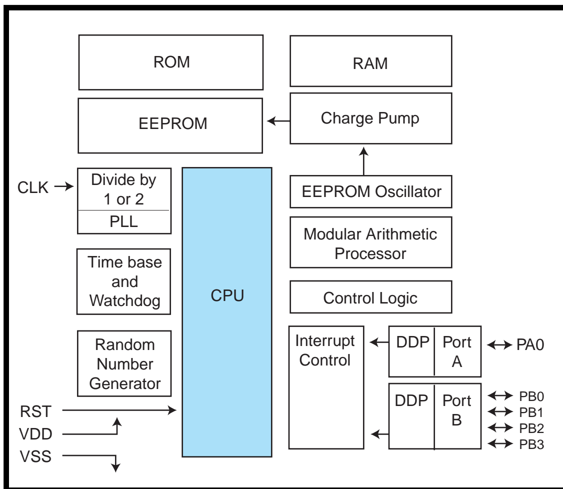


Figure 2—*At a minimum, the standard smartcard microcontroller contains a CPU and blocks of memory including RAM, ROM, and some sort of nonvolatile memory (usually EEPROM).*

Memory-management units are included on devices that support multi-application cards. Encryption and decryption hardware accelerators support additional algorithms with 1024 and larger key lengths.

Although it functions like a typical micro with instruction-set compatibility, the smart-card controller is fundamentally different because it's primarily designed for security. For instance, if you compare smart cards using the Motorola 68HC05 with its nonsmart-card Motorola counterparts, several differences become clear.

Most obvious is the smart card's single memory-mapped I/O. There are only five standard ISO-defined pinouts on a smart card: I/O, Clock, Power, Ground, Reset.

The smart card uses only onboard memory with relatively large amounts of nonvolatile memory. EEPROM programming is accomplished by an on-chip charge pump so it is controlled by the CPU and not accessible directly by external command. It appears stripped down compared to a nonsmart-card device since it contains no additional peripherals (e.g., ADC, PWMs, serial or parallel interfaces).

To increase mechanical robustness, smart-card devices are constrained by die size and use very dense memory elements. Devices for contactless cards use a microcontroller with analog circuitry that conditions the data and information transmitted over the interface. The card includes capacitive plates or a coil for coupling with the reader (see Figure 3).

MCU SECURITY FEATURES

Microcontrollers in smart cards strengthen system security. Security features can vary, but the aim is to restrict access to stored information and prevent the card from being used by unauthorized parties.

Each manufacturer includes some unique security features. These are never discussed to maintain security.

In general, however, a smart-card device includes one or more of these security features:

- multiple detectors for abnormal operating conditions keep the device operating in a well-characterized operating environment or forces a shutdown of the device or other protective action when unusual circumstances occur
- memory-mapped I/O under the control of the CPU restricts external access to the device
- controlled access to certain memory areas limits on-chip program modification of certain parameters
- unique serial numbers that track individual cards and security keys can be stored in protected memory
- on-chip oscillators and timers isolate the device from externally generated, potentially fraudulent clocks
- on-chip charge pumps program the EEPROM under CPU control

Manufacturers may also employ special tamper-resistant layers or nonstandard circuit and memory topologies. Test modes are isolated to ensure they can't be used to gain information and to ensure that the device executes only from its on-chip program.

THE ROLE OF SOFTWARE

Not only does smart-card software determine the card's functionality in the overall system, it also plays a huge part in the security of the system and the

card in particular. Typically, as much as 50% of a smart card microcontroller's ROM code is dedicated security.

The software takes advantage of the microcontroller's security features. It then verifies the different players by linking the terminal, card, and card carrier. Lastly, it encrypts and/or decrypts the data.

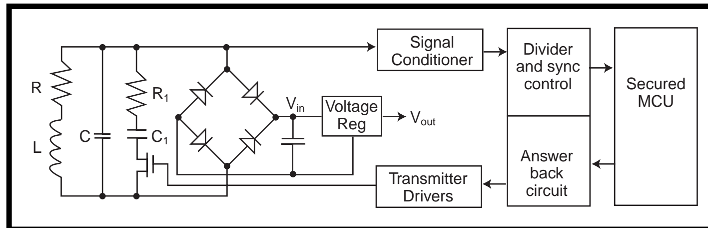


Figure 3—*Devices for contactless cards involve a microcontroller with analog circuitry that conditions the data and information transmitted over the interface while the card includes capacitive plates or a coil for coupling with the reader.*

To use the microcontroller security features, there are many aspects and techniques that you should consider—some running counter to what is normally considered good programming practice! The following suggestions are not all-inclusive or sufficient for all applications, but they point out the challenges of writing secure code.

Avoid implementing the application functionality using a standard method or routine. Make it unique so it's more difficult to understand or duplicate.

Deliberately use complex or illogical program flow. Alternate between having code in ROM and EEPROM to reduce the readability of the software. Periodically check that the EEPROM has not been erased or altered by verifying the state of some known, nonzero bits in the array.

Use variability in the software so a sensitive event does not always occur at a fixed time after reset. For example, use nonfunctional EEPROM writes to obscure the writes that are changing sensitive information. As well, use time-critical routines and the built-in watchdog function, and explicitly erase the entire RAM, or critical portions, whenever the device is reset.

You need to consider the consequences of atypical accidental or deliberate actions. What would happen if power is interrupted during critical portions of the software?

When you're determining state tables, don't assume that a set of inputs could not occur. What would happen if a set of conditions could be forced?

Check the state of any security flags (hardware or software) and/or registers before executing critical sections of code. When you implement a counter in software, ensure that the count is changed before the related action (e.g., comparing the input PIN) takes place.

And finally, limit the ability of the application to modify itself, but allow for controlled upgrade or modification of the system by the download of new software to the EEPROM.

The software controls not only the operation of the controller but also its relationship and interface to the rest of the system. There are usually three things that must be verified or linked before the flow of information and/or value can take place.

The system and its network (e.g., with the terminal) must be authenticated, and then the card is verified. This is often done by mutual authentication between the card and terminal via a handshake routine in which the card proves its validity to the terminal and the terminal to the card (see Figure 4). Optionally, you can link the card to the card carrier and verify that the card carrier is legitimate.

Currently, a PIN is commonly used to verify the cardholder. Unfortunately, it's not necessarily unique to the person and far too many people write their PIN on their card, nullifying any security benefits.

Various biometrics such as voiceprints, fingerprints, retina or iris scans, and dynamic signature patterns are being evaluated to provide the linking mechanism between the card and card carrier. The card carrier's unique biometric image captured by sensors at the terminal or card can be compared with the template stored on a smart card.

Such matching techniques are still imperfect and must prove acceptable by the card-carrying public. Depending on the application, designers must decide whether they're more interested in

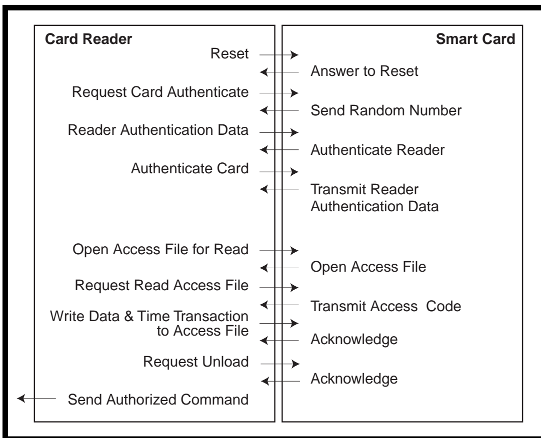


Figure 4—*Once mutual authentication between the card and the terminal is complete, an authorized action can take place.*

rejecting impostors or ensuring legitimate cardholders are always accepted.

Encryption and decryption algorithms are often used in smart-card systems. Cryptographic algorithms commonly maintain confidentiality, verifying the data's integrity, authenticating the sender's identity, or offering a way of nonrepudiation in a financial transaction. A variety of private key algorithms are available, such as DES (Data Encryption Standard), and public key algorithms like RSA (Rivest, Shamir, Adellman).

Most algorithms can be supported in software. However, system time constraints may necessitate an encryption coprocessor integrated into the smart card controller to accelerate performance of the algorithms. The smart card CPU controls the encryption coprocessor using various algorithm libraries.

Software can range from simple to much more complex algorithmic authentication processes. While restricting external access to the device, the software responds to an incorrectly entered PIN—anything from requesting a new attempt to completely locking up the device on a predetermined number of consecutive incorrect PIN inputs—or ask for additional information when behavior doesn't fit expected norms.

DEVELOPMENT ENVIRONMENT

The appropriate development environment hinges on the need for a proprietary versus standard solution (e.g., configurable off-the-shelf). You also have to take into account the developer's expertise, timeframe, and resources.

The traditional, most hands-on, resource-intensive, and time-consuming setup is to develop software using development tools specific to the smart card controller. This task usually requires assembly-code development.

However, tools specific to smart-card devices are often treated as confidential. They often require customer qualification before being provided (another security aspect that limits the access to smart-card information to those that have a need to know).

This method results in the most proprietary solution, is most tailored to the application's need, and is most efficient in code space. However, it restricts the available pool of knowledgeable developers and requires the longest development cycle.

You can also customize an available operating system to the specific application. Several companies offer OSs that target a particular application area and provide commonly used data structures, functionality, security techniques, and some customization.

These environments are provided with a higher level development tool and usually include the basics of a smart-card system (i.e., a handful of cards and a reader). This approach supports a faster time-to-market and requires less device-level expertise.

However, the resulting software may not be optimal. Tradeoffs often have to be made to fit within the customization options available, and the result is less efficient and proprietary code.

Advances in the smart-card controller also offer possible solutions. The advent of object-oriented software, such as Java interpretive code for smart cards, has radically changed the potential development environment.

You can develop an application applet using a commercial Java development environment and a smart card Java development kit, which contains a handful of cards with the interpretive code. The Java layer provides management and partitioning for a number of applications.

This approach minimizes time-to-market, considerably widens the field of potential developers, and results in a unique solution for the system. Most importantly, it supports the growing demand for multiapplication cards.

The major disadvantage is that, as a new technology, there are limited sources for the Java cards, and current smart-card microcontrollers are pushed to their performance limits by the demands of an interpreted layer. But, both of these concerns will rapidly change with new products coming to the market.

SMART CARDS AND SECURITY

By combining portability, computing power, and improved security, smart cards can be used in a growing number of applications. This proven technology has mature standards and a global-components infrastructure, so we're sure to see major technological advances in the next several years.

The biggest challenge to developing a smart-card system is also its greatest asset—security. Security pervades every level of the system, software, and chip design. ■

Carol Hovenga Fancher joined Motorola Semiconductor Products Sector in 1992 and is the America's Region Technical Marketer for the Smart Information Transfer Division. Prior to joining Motorola, Carol held engineering positions with Tracor, Ford Microelectronics, and Fraunhofer Institute for Integrated Circuits. You may reach her at r15544@email.sps.mot.com.

SOURCES

"Standards and Specifications of Smart Cards: An Overview"
Smart Card Forum
(703) 610-9023
Fax: (703) 610-9005
www.smartcardforum.org

ANSI (American National Standards Institute) and ISO (International Organization for Standardization)
(212) 642-4900
Fax: (212) 302-1286
www.ansi.org
www.iso.ch/welcome.html

ETSI (European Telecommunication Standards Institute)
+(33) 92 97 42 00
Fax: (33) 93 65 47 16

36 **Nouveau PC**
edited by Harv Weiner

40 **'*x*86 Processor Survey**
Part 2: '486-Class Embedded CPUs
Pascal Dornier

45 **Real-Time PC**
Network Communication
Ingo Cyliax

52 **Applied PCs**
A New View
Part 3: Sensors and Measurement Tools
Fred Eady

EMBEDDED PC

JULY 1998

RadiSys

Photo courtesy of RadiSys Corp.

LOGIC ANALYZER FOR NOTEBOOK COMPUTER

MobiLogic is a PC-based logic analyzer that operates through the enhanced parallel port of a computer. MobiLogic uses an external ISA chassis with two full slots, containing one or two PA600 PC-based logic analyzer cards installed, which gives the user a portable 400-MHz logic analyzer. The unit features transitional sampling, which greatly increases the effective memory depth by only storing data when one of the channels has changed. A time stamp is stored with every sample so that the time between samples is known.

MobiLogic is available in 48- and 96-channel versions with a standard memory depth of 64 Kb (as well as an optional 256 Kb) per channel. The case measures 3″ × 6″ × 15″ and weighs approximately 7 lbs. Its universal NC input power supply accommodates line voltages anywhere in the world. A parallel port cable connects the MobiLogic to either a notebook or desktop PC. The system operates under Windows NT or 95. Computer requirements are a ’486 or better with minimum of 8 MB of RAM.

Pricing for MobiLogic ranges from **$3995** to **$8995** depending on number of channels and memory. Current users of the PA600 can make their units portable by purchasing the empty MobiLogic for **$1000** and using it with their own laptop computer.

NCI
(256) 837-6667
Fax: (256) 837-5221
www.nci-usa.com

CompactFlash PC/104 MODULE

The **PCM-CFlash** is a small 3.6″ × 3.8″ (90 mm × 96 mm) low-cost adapter module designed to mount on a PC/104 stack. It links CompactFlash CF cards to the host computer through its ATA/IDE interface, which assures software compatibility. The module can replace conventional rotational disk memories in applications where floppy and hard disks cannot survive. It can also operate as the boot disk if no other disk is installed in the system.

The CF cards support both 3.3- and 5-V operation and can be interchanged between 3.3- and 5-V systems. CF cards include an intelligent microcontroller with an ATA/IDE interface so it appears as a standard IDE disk drive to the software. Virtually all OSs (including Windows 95, Windows CE, DOS, QNX, OS/9000, and Lynx) utilities and application programs support an IDE interface.

The PCM-CFlash is offered in an alternative configuration called the ADP-CFlash if PC/104 stack mounting is not desired. The ADP-CFlash permits remote mounting in a system to provide designers with maximum flexibility for accessing the CF card in their application.

The PCM-CFlash sells for **$59** (no CompactFlash card supplied).

WinSystems, Inc.
(817) 274-7553
Fax: (817) 548-1358
www.winsystems.com

NouveauPC

edited by Harv Weiner

233-MHZ SBC

The **VIPer821** is a fully integrated 233-MHz MMX industrial single-board computer that features full desktop/workstation functionality packaged into a half-size ISA-bus form factor. The computer supports PC/104, ISA-bus, and stand-alone operation. Onboard features include 256 MB of DRAM, 512 KB of L2 synchronous cache, onboard PCI video, PCI 10/l00Base-TX Ethernet, CompactFlash card technology, and USB support. A 64-bit PCI graphics engine delivers high-resolution video to 1280 × 1024 × 256 using 2 MB of EDO video memory. Both flat panels and CRTs are supported, and V-port compatibility enables real-time video as well as graphics over video overlays.

CompactFlash technology currently permits the addition of up to 24 MB of user-upgradable flash memory to the VIPer821. Also, because its interface is IDE-compatible, virtually any operating system can access—or even boot from—the flash card without requiring a special BIOS or driver. Flash memory is critical for storing data in mobile and data-collection applications.

The VIPer821 is fully integrated with all the standard I/O, including support for serial and parallel ports, hard disks, floppy disks, and USB. Supervisor circuitry includes a watchdog timer, power-fail/low-battery detection, and a CPU-temperature sensor/alarm. The board also supports advanced power management and features a CPU-temperature monitoring and control algorithm. The VIPer821 is compatible with all popular operating systems including MS-DOS, Windows 95 and NT, OS/2 Warp, QNX, SCO Unix, and Novell.

The VIPer821, with an Intel 133-MHz microprocessor, PCI 10/100Base-TX Ethernet, 2-MB EDO video memory, and 512-KB L2 cache installed (but no memory) sells for **$1640**.

Teknor Industrial Computers, Inc.
(800) 387-4222
(561) 883-6191
Fax: (561) 883-6690
www.teknor.com

SCALABLE FLASH DISK DRIVE

The **MD104** scalable flash disk drive combines up to eight DiskOnChip2000 drives on a single board. The DiskOnChip2000 devices range in storage capacity from 2 to 40 MB, enabling MD104 configurations with capacities from 2 to 300 MB on a single PC/104 card. The anticipated 72-MB DiskOnChip2000 devices will expand the capacity to more than 500 MB.

The MD104 can be used for embedded applications in harsh environments such as aviation, robotics, and vehicle-mounted computers. Also, its plug-and-play capability frees you from needing external software drivers, as well as providing 100% hard-disk emulation and compatibility.

The MD104 is priced at **$99**, not including the M-Systems DiskOnChip2000, which presently costs between **$12** and **$22** per megabyte, depending on the device capacity chosen.

Tri-M Systems Inc.
(604) 527-1100
Fax: (604) 527-1110
www.tri-m.com

PCI ADC WITH ONBOARD CPU

The **PD-MF-330/12** family of A/D boards consists of four models with a "processor based" Motorola 56301 PCI DSP interface. These boards let the user offload the host CPU data-acquisition functions to the onboard DSP, thus giving the user two CPUs in one PC.

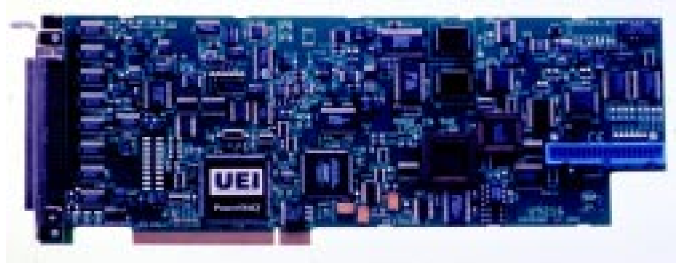
Each board features four subsystems—analog input, analog output, digital I/O, and counter/timers. The analog input features 16/8 or 64/32 channels, 12-bit resolution, 1-KB FIFO, 330-kHz throughput, and programmable gains of 1, 2, 4, 8 or 1, 10, 100, 1000. The analog output consists of two 12-bit 200-kHz per channel DACs. The digital I/O consists of eight digital-input lines, which can generate interrupts, and eight digital-output lines. Three 8254-type counter/timers are available.

The PowerDAQ technology lets all the subsystems run simultaneously or independently with one or more boards in the same PC. Multiple subsystems can be started or stopped as required. All of these A/D boards feature extensive hardware and software triggering. Additionally, the data-transfer methods include slave-mode and bus-mastering operation.

PowerDAQ software for Windows 95 and NT features "clean" 32-bit code, full event-driven multithreading support, and source code for the DLL. The software supports Visual C++, Visual BASIC, LabVIEW for Windows, LabWindows CVI, TestPoint, and HP VEE.

The 330-kHz 16-channel board is priced at **$895**, and the 64-channel version costs **$1395**.

United Electronic Industries, Inc.
(800) 829-4632 • (617) 924-1155
Fax: (617) 924-1441
www.ueidaq.com



NouveauPC

CE-READY BIOS

Embedded BIOS 4.1 initializes an industrial PC or embedded target as it would for DOS, Windows 95, or Windows NT, and then boots Windows CE from ROM, flash memory, or disk. This new capability—CE Ready—enables industrial PCs and targets using Embedded BIOS to boot the entire range of industry-standard OSs. No ad hoc third-party loader or launcher software is required, nor is DOS required, to boot Windows CE in a CE Ready system. This innovation keeps the system software simple and standard, while reducing costs by eliminating DOS and loader royalties in a Windows CE system.

Embedded BIOS 4.1 has a built-in debugger for checking out address and data paths, memory, and flash-memory components. The manufacturing mode enables the system's disk drives and flash-memory disks to be managed remotely and permits the BIOS or Windows CE to be reflashed in the system over a standard RS-232 connection to a host PC. Remote console redirection lets you redirect traditional keyboard and screen I/O over an RS-232 connection to a host PC running terminal-emulation software, so you can debug targets without keyboards or screens.

Embedded BIOS 4.1 comes with royalty-free copies of Embedded DOS-ROM and its Resident Flash Disk software, which emulates floppy or hard disks with solid-state flash media.

General Software, Inc.
(800) 850-5755
(425) 454-5755
Fax: (425) 454-5744
www.gensw.com

NouveauPC

Pascal Dornier

'x86 Processor Survey

Part 2: '486-Class Embedded CPUs

Last month, Pascal compared '386 CPUs so you'd have a feel for which one would suit your design specs. However, there are times you need the extra oomph of a '486 machine. Find out whose CPU best meets your needs.

How do you pick a CPU?

First priority: good software development and operating environment. A cheap CPU isn't much good if you then waste a lot of time and money dealing with weird tools or operating systems.

Evaluation boards are important, too. You can configure them to closely resemble your target configuration, and test your software in parallel with hardware development. Many embedded applications require long-term availability, which may rule out some parts that mainly target the consumer market.

Just as important is a commitment to engineering support. CPU performance, feature set, power, and PCB real estate have to match your application and budget.

BIOS adaptation is another consideration. The more configuration options a CPU has, the more time it will take to configure and debug everything. Once you consider all of these issues, it shouldn't be hard to make a decision.

In Part 1, I covered '386-class embedded CPUs. The '486-class CPUs I discuss this month build on the experience gained from these earlier products. You'll see how the '486s are usually quite an improvement over their predecessors.

AMD ÉLAN SC400

The AMD Élan SC400 was designed to provide a fully integrated solution for hand-held devices such as PDAs and wireless terminals. Design wins include a Windows terminal, a portable navigation system, and several Web-browser and set-top box designs.

The CPU core is based on AMD's '486 and runs at up to 100 MHz. For lowest power consumption, the core voltage can be reduced down to 2.7 V.

As depicted in Figure 1, integrated peripherals include a LCD controller (frame buffer stored in the first 16 MB of main

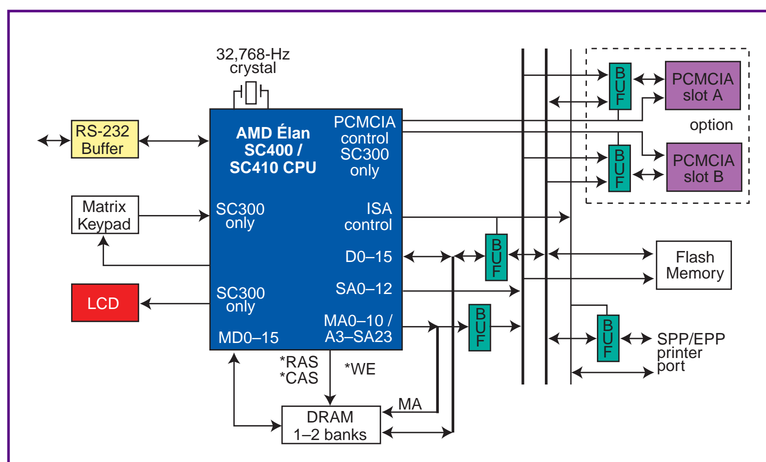


Figure 1—Like its predecessor, the SC300, AMD's Élan SC400 includes all the functions you require in a hand-held computer.

memory), a PCMCIA controller, one serial and one parallel port, a real-time clock, and a XT keyboard interface. Instead of an external keyboard, a keyboard matrix can be scanned by software, and the 8042 interface emulated through *SMI or NMI routines.

As you'd expect from a part designed for mobile use, the Élan SC400 has a sophisticated power-management unit (PMU). Power management can run fully in hardware, without software intervention, or as a combination of the PMU and NMI- or *SMI-based software. The digital I/O pins can be programmed to change based on the power state. With the right peripherals, the PMU could be set up to brew coffee, bake croissants, wake up the CPU….

One hand gives, the other takes away. Not all features are available at the same time, owing to pin count and other limitations. One of the "features" is that the LCD controller can only be used with 16-bit DRAM, not with 32 bit (just when more bandwidth is needed).

Most pins are 5-V tolerant. Unfortunately, one half of the DRAM bus is not, which requires either 3.3-V DRAM or level shifters.

AMD's Élan SC410 is a lower cost, stripped-down version of the Élan SC400. This device has the LCD and PCMCIA controllers removed.

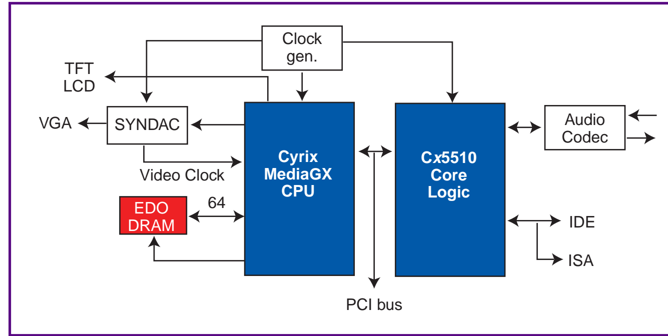


Figure 2—With few additional components, you can build a complete multimedia PC around the Cyrix GX chipset.

CYRIX MediaGX

The Cyrix GX CPU was originally designed for low-cost PCs and notebooks. So far, it has been designed mainly into low-end desktop PCs, such as Compaq's Presario 2100 and 2200 models, but some recent design wins include notebooks like the Compaq Presario 1220. The MediaGX is also designed into COM1's SurfTV set-top box.

The GX is a two-chip solution, as you see in Figure 2. The main chip includes the CPU core, a simple CRT controller and graphics accelerator, and a fast 64-bit memory controller. It connects to the companion chip (Cx5510) through the PCI bus.

The companion chip includes the PCI-to-ISA bridge, the usual ISA-bus peripherals, the PMU, and some Sound Blaster emulation logic. The idea behind this

Power Table

Part	+5V	Gnd
U1	2,12,24,39 72,98,113,124	9,21,37,69 95,110,121,131
U2	32	16
U3	4	2
U4	1,21	22,42

Figure 3—The National NS486SXL can be used like a microcontroller in this minimal configuration. (This design has not been tested.)

partitioning is to minimize the die size of the CPU chip (built in a relatively expensive high-performance process) and use lower cost generic technology for the companion chip.

The GX is based on a fast 5x86 core, running at up to 233 MHz. The high speed enabled Cyrix to dispense with a lot of legacy hardware.

VGA and Sound Blaster functions are emulated through SMI interrupts. Cyrix calls this Virtual System Architecture (VSA). The native Windows drivers access the linear frame buffer directly and don't incur any emulation overhead.

To reduce cost and pin count, the main memory and graphics frame buffer are unified. To minimize the memory bandwidth required for screen refresh, frame buffer data is automatically compressed in a shadow frame buffer if possible.

The EDO memory controller integrated in the CPU runs at the same frequency as the CPU core. Therefore, DRAM timing can be optimized with a very fine granularity.

To minimize DRAM page misses, the GX CPU lets the user set aside a portion of the cache as a scratchpad. Special instructions can copy DRAM data to the scratchpad. The bit block transfer engine can copy this data to the frame buffer. Part of the scratchpad is also used by the SMI handlers to reduce SMI overhead.

An external RAMDAC/clock generator is required to drive a CRT. The GX can directly drive TFT LCD panels. Unfortunately, it can't drive DSTN panels, which would have been more appropriate for a low-end notebook product. Because of this, the Compaq Presario 1220 notebook disables the on-chip video controller and uses an external controller instead.

Sound input and output can be supported by connecting a audio codec to the Cx5510. This solution can be quite cost effective for simple wave audio output. If sound synthesis is required, I recommend using one of the many Sound Blaster–compatible controllers. Most likely, they'll cost less than the CPU performance required to perform the emulation.

A new version of the companion chip (the Gx5520) includes the RAMDAC and audio codec on chip. What's the catch? It's the most expensive CPU in the group. Also, I've had a lot of difficulty getting detailed information and support for this part. I hope the merger of National and Cyrix will improve this situation.

	AMD SC400 (*SC410)	Cyrix MediaGX	National NS486SXF(*NS486SXL)	SGS STPC Consumer
CPU core	AMD '486	Cyrix 5x86	National '486SX	Cyrix '486
CPU frequency	33/66/100 MHz	166–233 MHz	25 MHz	66–133 MHz
CPU cache	8-KB code/data	16-KB code/data	1-KB code only	8-KB code/data
Input clocks	32,768-Hz crystal	60–66 MHz, video clock	50-MHz crystal, 32,768-Hz crystal	14.318-MHz crystal
Coprocessor	not supported	internal	not supported	internal
Maximum power	1.5 W (3.3 V, 66 MHz) 0.94 W (2.7 V, 66 MHz)	5.8 W (2.9 V, 180 MHz)	1.0 W(*0.8W)	3.2 W (3.3 V, 66 MHz)
Voltage range	3.0–3.6 V (CPU core can run down to 2.7 V)	3.0–3.6-V I/O, 2.9-V core	4.75–5.25V	3.0–3.6V
Idle power	63 mW standby 0.17 mW suspend	1.3 W standby 29 mW suspend	40 mW standby 0.1 mW suspend	TBD
Temperature range	0–70°C Tambient	0–70°C Tcase	0–70°C Tambient	–40–85°C Tambient
Price	$54.16 (*$49.12)@ 100–1000 (Hallmark)	$45 (OEM, 180 MHz, 5510/5520 extra)	$33.70 @ 100(*$25.30) (Hallmark)	$40 (10k OEM)
Package	292 BGA	352 BGA + 208 QFP	160 QFP(*132)	388 BGA
DRAM support	16 or 32 bit, FPM or EDO, up to 64 MB	64 bit, EDO or SDRAM, up to 128 MB	16 bit, fast page mode or EDO, up to 16 MB	64 or 32 bit, EDO, up to 128 MB; 32-bit no VGA
Timer channels	3 (8254)	3 (8254)	3 (8254)	3 (8254)
Watchdog	no	no	yes	no
Interrupts	2 × 8259 + *SMI	2 × 8259 + *SMI	2 × 8259 + NMI	2 × 8259
DMA channels	7 (2 × 8237)	7 (2 × 8237)	4 channels(*none (nonstandard) + ECP)	7 (2 × 8237)
Keyboard interface	XT interface, matrix scan	no	no	no
RTC	yes	no	yes	no
Serial port	1 × 16550 with IrDA	no	1 × 16550 with IrDA; Microwire/Access.bus	no
Parallel port	1 × SPP/EPP (requires buffer + latch)	no	1 × SPP/ECP(*no)	no
Programmable chip selects	up to 15	no	up to 29(*up to 28)	1
LCD/CRT controller	yes, monochrome/color LCD (up to 16 colors/gray shades max. 128-KB frame buffer)(*no)	CRT (external DAC) and TFT support (linear frame buffer with 2D graphics accelerator)	up to 480x320 LCD,(*no) 4 gray shades	CRT and TV out, integrated RAMDAC, 2D graphics accelerator, VGA compatible, video input
PCMCIA	2 sockets, 82365 compatible(*no)	no	1 socket, 82365 compatible(*no)	no
IDE interface	use programmable *CS	yes, PCI bus master	use programmable *CS	PCI bus master
Bus interface	ISA, VL optional	PCI, ISA	ISA	PCI, ISA
Bus masters	not supported	PCI, ISA	ISA, needs buffers	PCI, ISA
5-V tolerant I/O	partial	yes	yes	yes
PC compatible	yes	yes, using SMI	no	yes
Testability	JTAG	no	limited pin scan	no

Table 1—Take your pick among these '486-class CPUs—fast, cheap, or low power (you can't have it all).

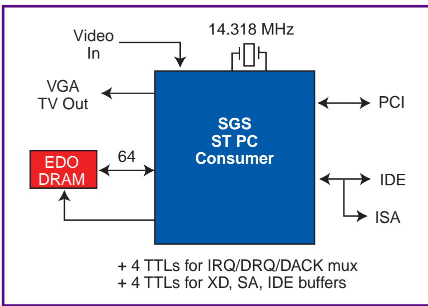


Figure 4—The SGS ST PC Consumer was designed for set-top boxes, so it includes video-in and TV-out functions.

NATIONAL NS486SXF

The National NS486SXF CPU was designed for real-time operating systems and is only partially PC compatible. The CPU core is stripped down and doesn't support real-mode or virtual memory paging.

The NS486SXF includes a complete set of peripherals—a serial and parallel port, a gray-scale LCD controller, a PCMCIA controller, and a Microwire/Access.bus (I²C) interface. Also, one of the timer channels can be reprogrammed as a watchdog timer.

The 25-MHz clock speed and small 1-KB cache size mean this CPU is the slowest in this group. However, its low price and integrated peripherals still make it attractive for applications such as simple industrial user interface/control functions or small network appliances like print servers. This CPU comes closest to the simplicity of a microcontroller, as Figure 3 illustrates.

Since the DRAM address lines are multiplexed with the ISA address bus, ISA-bus mastering requires external buffers. The NS486SXL is a stripped-down version of the NS486SXF, with the LCD and PCMCIA controllers removed.

The NS486SXF runs at 5 V, so power dissipation is rather high relative to CPU performance. The clock can be driven either by a 50-MHz fundamental crystal (hard to get) or a crystal oscillator.

SGS-THOMSON ST PC CONSUMER

The ST PC Consumer CPU diagrammed in Figure 4 is designed primarily for TV set-top box and consumer PC applications. However, its cost should also be competitive for many embedded applications that don't require the video support it offers.

The CPU is based on the '486 core licensed from Cyrix, running at up to 133 MHz. All clock generators are built in. The only thing required is a 14.318-MHz crystal.

The integrated video controller is VGA compatible with an internal RAMDAC and TV out (NTSC/PAL) encoder. The frame buffer is stored in main memory. The ST PC Consumer has a video-input port as well as a video-output pipeline (scaler, chroma, and color key).

The PCI-to-ISA bridge is also built in. To reduce the pin count, it uses external multiplexers for the ISA-bus IRQ, DRQ, and DACK pins.

BGA PACKAGES

Most new embedded CPUs use ball grid array (BGA) packages. Many embedded designers are concerned about this since BGA packages can't be soldered or desoldered reliably without expensive hot-air tools (starting around $3000). Also, most pins of a BGA package cannot be inspected visually, but only by x-ray.

I recommend a pad size of 24 mil and a solder mask opening of 28 mil. Two traces can be routed between each pin by using 5-mil traces and spaces. Outside of the BGA area, the traces can expand to a more conventional 6- or 8-mil design rule.

EPC

For reliable results, have your prototypes professionally assembled using a solder paste stencil. It costs more than hand assembly, but it's worth it.

X-ray inspection is of limited use. It can spot shorts but not opens. Shorts can be avoided through clean processing.

Remember that BGA packages are moisture sensitive (popcorn effect). Don't open the sealed bag until the parts are to be soldered, or you'll have to bake them according to the manufacturer's recommendations to remove moisture from the package.

BGA packages should be rugged enough for most applications, but they may be problematic when they encounter frequent temperature extremes. The solder balls ensure a large distance between the component and the board, and they enable much easier flux removal than most other surface-mount packages. With a good process, BGA assembly yield should be close to 100%.

IN THE CRYSTAL BALL

In the second half of 1998, Cyrix plans to introduce the MXi integrated processor. The MXi is based on Cyrix's next-generation CPU core with a 64-KB cache. Thanks to a 128-bit SDRAM interface, memory bandwidth will be up to 2 GBps, which should enable MXi to provide high 2D and 3D graphics performance.

SGS-Thomson is also working on other versions, including ST PC Industrial, which adds serial and parallel ports, PCMCIA/Cardbus, and TFT support and deletes the IDE, TV-out, and video-in functions. They also offer customer-specific configurations for a substantial volume commitment.

And what about Intel? I didn't cover its embedded '486 and Pentium products because they don't integrate the core logic on chip. The embedded Pentium module might be interesting for some applications. However, the Pentium Pro is being phased out rapidly, and the packaging of the Pentium II is a poor fit for embedded designs.

WHICH CPU FOR YOU?

Table 1 lays it all out. If your product runs off batteries or if high integration is essential, the AMD Élan SC400 series is the most likely fit. The National NS486SXF and NS486-SXL are best for cost-sensitive applications where the designer fully controls the software.

The SGS-Thomson ST PC Consumer and the Cyrix MediaGX are both strong contenders for set-top box designs. They also do well for applications needing high CPU or PCI expansion bus bandwidth, such as networking.

Good luck finding the best '486 fit for your design. EPC

Pascal Dornier is president of PC Engines, a design house for embedded PC hardware and firmware. You can reach him at pdornier@pcengines.com.

SOURCES

Élan SC400, Élan SC410
Advanced Micro Devices, Inc.
(800)538-8450
(408) 732-2400
www.amd.com

MediaGX
Cyrix Corp.
(800) 462-9749
(972) 968-8388
Fax: (972) 699-9857
www.cyrix.com

NS486SXF, NS486SXL
National Semiconductor Corp.
(800) 272-9959
(408) 721-5000
Fax: (408) 746-3096
www.ns486.com

ST PC Consumer
SGS-Thomson Microelectronics
(617) 259-0300
www.st.com

Real-Time PC

Ingo Cyliax

Network Communication

With the advent of cost-effective TCP/IP, gone are the days of writing your own communication protocols. Ingo shows how easy it is to set up a real-time network whether you choose to use a serial connection or the Ethernet.

I enjoy getting out to the embedded-systems conferences. For one thing, it gives me the chance to scrounge around for more hardware and software for this column.

But, it also gives me an opportunity to communicate and network, as it were, with fellow engineers. In talking to you, I get feedback, I get ideas, I get to know you better.

So this month, let's communicate some more. In this context, though, I'm talking about a different kind of communication.

It used to be that embedded-systems engineers devised their own protocols so their systems could communicate with each other. These days, however, we like to reuse as much as possible, and this trait is clearly evident in the communications area.

Writing your own protocol, although it seems easy at the beginning, is not so simple. I started out thinking this column would be about implementing protocols (e.g., frames, checksums, and sliding-window protocols). But in fact, it's much more cost effective to use TCP/IP for many applications.

If you don't want to take my word for it and still want to implement your own protocol, check the networking books in the reference section. They go through excruciating pains to show you how to implement a sliding-window protocol with checksums and so on.

Before we get into TCP/IP, let's look at some communications hardware. There really isn't that much common hardware out there. There are serial ports such as RS-232 and RS-422/-485 and network interfaces like Ethernet.

There are device buses like CAN and so forth, too. But, these are application specific, and they're complex enough to require their own column.

After we take a look at these, I'll check out TCP/IP and show you an example that illustrates how simple it can be.

RS-232

You're probably most familiar with RS-232. It has been around for, well, forever. Every PC has usually at least two RS-232 ports. They're standard.

RS-232 only defines the electrical specification of the interface and the bit-level coding. What you send isn't defined by the standard, so it's up to the application.

However, RS-232 has some limitations. The data rate is typically limited to less than 115 kbps, and the maximum distance you can effectively run the connection is quite limited (50′ max., less at 115 kbps).

Also, the signaling isn't isolated. It uses a common ground between the devices. This setup can be a problem, especially in high-noise environments (e.g., factory floors). The noise-immunity and length limitations can be improved by using line drivers and modems that translate the RS-232 signaling to a different electrical, or sometimes even optical, signaling method.

But even with line drivers and modems, RS-232 is a point-to-point communication standard. To hook up an RS-232–based device, you need a pair of serial ports—one on each device for a link. Also, PC architectures are usually limited to four USARTs (COM ports) before you have to do something special.

RS-422/-485

RS-422/-485 addresses some of the shortcomings of RS-232. It uses differential signaling to improve noise immunity and distance (up to 4000′). RS-422/-485 also runs faster, typically into the 1-Mbps range.

Besides this, RS-485 allows multidrop connections. So, more than two devices can be attached to an RS-485 bus.

But, like RS-232, the RS-422/-485 standard doesn't define the protocol for speaking over the link. To implement a multidrop network, the application must implement an arbitration scheme to be used by the device on the network.

Also, since the electrical interface is more sophisticated, RS-422/-485 cards are generally much more expensive than RS-232 cards.

ETHERNET

Although device buses like CAN and LonWorks exist, controllers for Ethernet are becoming quite inexpensive. Therefore, some implementations use Ethernet as a device bus for communication between controllers, acquisition devices, and other systems.

Ethernet runs on a variety of physical media, such as twisted-pair cables, coax, and fiber. In other words, it's suitable for many applications and environments.

Also, Ethernet is available in two speeds—10 and 100 Mbps. Faster, gigabit Ethernet implementations are expected on the market soon.

Since cards for Ethernet are driven mostly by the desktop market, they have really dropped in price. You can find Ethernet cards for less than $20, and chips that can be embedded on a motherboard or application are available for under $10. Since the low cost of Ethernet equipment is mostly due to consumer applications, components are available from many sources.

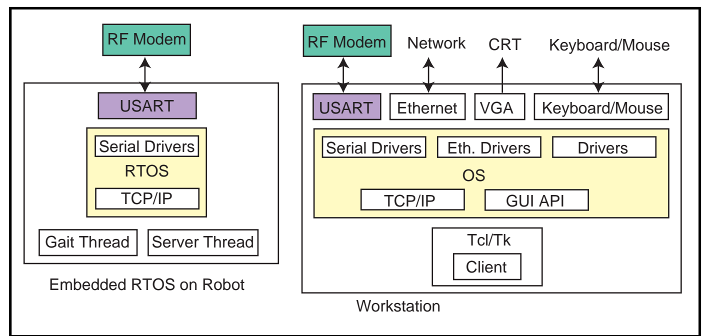


Figure 1—Using a point-to-point link over a wireless modem, you can control a robot running a real-time application. The user interface runs on a notebook but could run on any system supporting Tcl that has a network interface.

And because Ethernet is a standard, many systems speak it. It's possible to build real-time–based systems that can communicate with desktop systems and nonreal-time servers without adding extra software.

Sometimes, however, you may want to use Ethernet for a real-time application. But, it's inherently nondeterministic. Every time a node connected to the Ethernet wants to transmit data, it has to wait to make sure there's no traffic before sending its data.

Also, it does not know if another Ethernet device is doing the same thing at the same time. Occasionally, two devices simultaneously decide the network is free, so they transmit at the same time, resulting in a collision.

Of course, there is a mechanism for dealing with this, but as a result, Ethernet can't guarantee deterministic delivery of data. This effect gets worse the more nodes and traffic are on the Ethernet segment.

The most common method for getting around this situation is to make sure the Ethernet segment is not very loaded. In fact, if there are only two nodes on the segment, then performance is predictable.

To increase utilization, you can use faster than necessary Ethernet implementations. In other words, using 100-Mbps Ethernet instead of 10 Mbps makes it 10 times less likely to be congested for the same traffic.

Finally, you can implement pseudo-token-passing or polling protocols over Ethernet. This technique introduces a lot of overhead but makes the network deterministic if every node participates in the algorithm.

ETHERNET PROTOCOLS

The most common protocol used over Ethernet is TCP/IP. But, TCP/IP is kind of a misleading term, so let's discuss it a bit.

TCP/IP can mean the suite of protocols used in IP implementations (e.g., TCP/IP stack) or the combination of the terminal control protocol (TCP) over Internet protocol (IP), which implements a reliable peer-to-peer windowing protocol, commonly referred to as a stream. Most of the time, the protocol-suite usage is implied because stream-based connections are the typical communication mechanism.

IP, which is the foundation of the Internet-protocol suite, refers to the layer responsible for switching packets between networks and addressing individual nodes in a network. The IP layer is implemented on top of Ethernet frames, which is the basic packet type implemented by Ethernet controllers. The IP header addresses nodes using an Internet address, which is a 32-bit number represented as a dotted quad (e.g., 193.76.43.1).

For applications to use IP, they need to be able to address specific resources within a node. This task is accomplished with a host address and a port number (introduced by UDP and TCP). Besides enabling you to

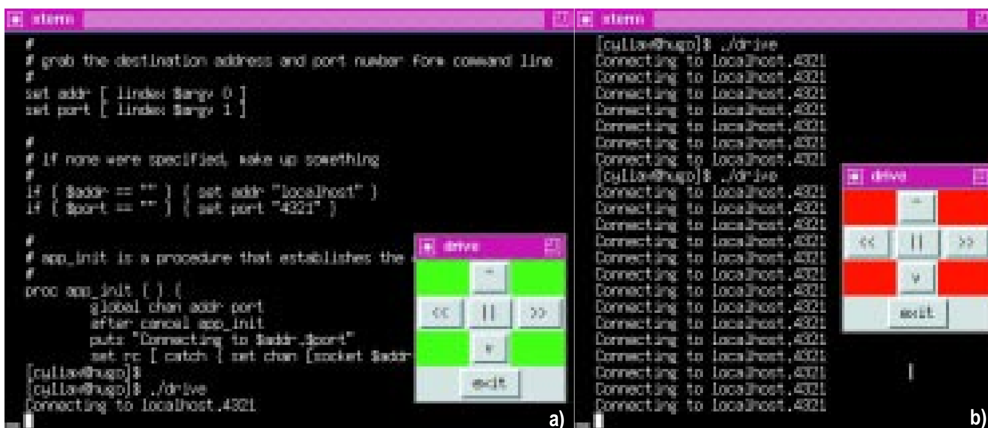


Photo 1—In the GUI for the client application, each direction has a button. When the user presses the button, the program sends a command to the robot. a—A green background indicates an established connection. Although this was running on my notebook under Linux, it would look similar under Windows. b—The red background indicates the connection to the robot was lost. The program tries to reinitiate the connection until it succeeds.

address individual resources on a node, TCP implements a connection-oriented communication path between two resources.

In Internet terminology, sockets are communication endpoints and streams (in the case of TCP) or data grams (in the case of UDP). The API commonly used to write applications employing these protocols is called the Socket API. Socket API is implemented in almost all OSs these days, including many PC-targeted RTOSs.

TCP/IP offers other advantages as well. For one, it is network-interface independent, so with appropriate network drivers, you can use the same protocol to communicate over Ethernet, serial (point-to-point or multidrop), and wireless communication channels. The application using Socket API to interface to the TCP/IP stack does not care (or even know, in most cases) what kind of link-layer communication channel is used to get off the node.

TCP/IP OVER SHARED MEMORY

Since TCP/IP insulates the application from the link-layer communication chan-

Listing 1—After the main thread of the server process spawns off the gait generator, it initializes the socket it uses to listen to connections. Once a connection is received, the `accept()` call will unblock and the thread enters the command interpreter loop. This particular server only allows one connection to be active at a time.

```
#include <netinet/in.h>
#include <stdio.h>
#include "global.h"
#include "servo.h"
#include "gait.h"

int cmdmutex;
int servomutex;
extern int Command;
main(){          /* main thread: init system, start GaitThread,  */
int s,ns;        /*   set up TCP/IP port, accept connections and */
struct sockaddr_in sin;      /*   do command loop */
int slen;
int GaitThread();
int n;
char buf[256];
  SpawnThread(GaitThread);
  servomutex = CreateMutex();
  cmdmutex = CreateMutex();
  s = socket(AF_INET, SOCK_STREAM, 0);   /* cmd port via TCP/IP */
  sin.sin_addr.s_addr = htonl(MyIPAddress);
  sin.sin_port = htons(MyPort);
  bind(s,(struct sockaddr *)&sin,sizeof(sin));
  listen(s,1);
  while(1){                  /* main loop */
    slen = sizeof(sin);      /* wait for connection */
    ns = accept(s,(struct sockaddr *)&sin,&slen);

#ifdef DEBUG
    fprintf(stderr,"Connect from %s.%d\n",
            inet_ntoa(sin.sin_addr),
            ntohs(sin.sin_port));
#endif
    while(1){                /* do command loop */
      if((n = ReadLine(ns,buf,sizeof(buf))) < 1)
        break;
      GetMutex(cmdmutex);
      Command = DecodeCommand(buf);
      ReleaseMutex(cmdmutex);}
#ifdef DEBUG
    fprintf(stderr,"%s.%d Disconnected.\n",
            inet_ntoa(sin.sin_addr),
            ntohs(sin.sin_port));
#endif
    GetMutex(cmdmutex);  /* remote process has closed connection */
    Command = CMD_STOP;
    ReleaseMutex(cmdmutex);
    close(ns);}}
```

nel, you can also use it to communicate with processes on the same system. In a sense, the TCP/IP stack can be used as an interprocess communication protocol.

Several schemes, such as the remote procedure call (RPC) and network file system (NFS) protocols, use this feature. Applications can then communicate with each other whether they are on the same or different processors.

Multiprocessor systems can also use the TCP/IP stack and the Socket API to communicate via shared memory over buses. This is particularly interesting in multimaster bus systems like VME or PCI. You can even build distributed systems using shared-memory communication by connecting crates, a backplane with boards, or bus-repeaters over fiber or other links.

The distinction between shared-memory distributed systems and traditional network-based systems becomes a little fuzzy when you use the Socket API and TCP/IP to communicate between processes. I'm mentioning it because it points out how flexible and widespread the use of TCP/IP and Socket API has become.

VENDOR SUPPORT

So, what's available from vendors? As I mentioned, just about any PC RTOS has TCP/IP support these days. About the only difference is the network interface device they have support drivers for.

One fairly common protocol, SLIP for asynchronous serial application, uses standard PC-architecture UARTs over RS-232 and modems. Point-to-point protocol (PPP) support is also available for many systems and is quickly becoming the standard serial-based IP for many Internet service providers.

Ethernet support is a little trickier. Most RTOSs support NE2000-compatible Ethernet cards, but many kinds of Ethernet cards are on the market. Pick one that fits your budget and is supported by your RTOS.

Writing an Ethernet driver is not for the faint of heart, but it can be done if the card vendors give you programming information for the card. Several RTOS vendors have some sample drivers available in source form, which is a good starting place.

Since Ethernet is becoming a de facto device bus, suitable for many applications, Ethernet acquisition devices are becoming quite cost effective. Keithley features a line of acquisition devices with Ethernet interfaces. Expect more to come from others.

I'm still waiting for inexpensive Stamp-like devices with an Ethernet interface. A real-time system could use these for remote data collection and control. One of the **Design98** winners implemented components of the TCP/IP stack on a PIC (see "PIC of the Lot," *INK* 95), so such devices aren't far off.

ROBO CHAT

By using the TCP/IP suite, I can write applications independent of the communication channel used. If your OS implements the Socket API, your application is also insensitive to the OS used.

Let's look at the communication system of the robot controller I introduced in *INK* 92. This controller uses an RTOS to manage several tasks.

The robot is legged, so the system has to control the gait (i.e., the sequence of leg activations) to make the robot move. To drive the actuators (hobby servos commonly used for radio-control models), the system needs to generate 12 channels of programmable pulses that lie within 1.0 and 2.5 ms under program control. Clearly, I'm dealing with hard real time.

Finally, the system has a communication channel to an external controller, which is responsible for the high-level control behavior (e.g., move forward and backward, turn). The external controller is a workstation, like an NT or Unix machine, that has a network connection as well as a serial port to use for the robot communication.

The communication channel between the robot and workstation is RS-232 and uses wireless modems to achieve about 19.2 kbps. To ease system programming and make the robots accessible from anywhere, the RS-232 channel runs SLIP to enable it to carry TCP/IP traffic.

Figure 1 illustrates how simple it is to write a TCP/IP application that communicates over a network where SLIP and the TCP/IP stack are already implemented on both the workstation and the robot's RTOS. The instructions for installing and configuring the networking code is standard and you can read the vendor's literature, so I won't cover it.

There are two programs I need to look at—the TCP/IP server application, which runs on the robot, and the TCP/IP client application, which runs on the workstations. The terms "server" and "client" describe the semantics of the programs.

In IP speak, a server is simply a program that listens for connections from a client program and starts communicating. Since the server program usually keeps running after the client disconnects, it is sometimes also called a server demon process.

My server program (see Listing 1) is a simple loop that listens for a connection using the `accept()` call. `accept()` blocks until a client connects. When this happens, the RTOS fills in the `sin` structure with the identity, address, and port number of the client socket and returns a new file handle to use for the life of the connection.

The server then enters the command loop, where it waits for data from the client and sets the current command mode for the gait controller. The commands are simple, like `CMD_FORW` and `CMD_STOP`.

When the client is done, it initiates a close on the connection. `ReadLine()` returns a zero, indicating the connection is gone. The program then closes the connection and returns to listening for new connections.

The client program is written in task control language (Tcl) and runs on my notebook, which runs Linux for this project. Tcl is an interpreted language, which enables you

Listing 2—Here is the client program, which implements both the GUI and command generator. The program is written in Tcl, which is a portable language available for MacOS, Unix/Linux, and Windows for free.

```
#! /usr/bin/wish
# grab destination address and port number form command line
set addr [lindex $argv 0]
set port [lindex $argv 1]

# if none were specified, make up something
if {$addr == ""}{set addr "localhost"}
if {$port == ""}{set port "4321"}

# app_init establishes connection
proc app_init { }{
  global chan addr port
  after cancel app_init
  puts "Connecting to $addr.$port"
  set rc [catch {set chan [socket $addr $port]}]
  if {$rc == 0}{
    .top configure -bg "green"}
  else{
    set chan 0
    after 1000 app_init}}

# call app_done when ready to tear down connection
proc app_done { }{
  global chan
  if {$chan != 0}{
    close $chan
    set chan 0
    .top configure -bg "red"}
  exit}

# app_cmd is command button callback for sending command via
# communication channel to robot. If error occurs, close channel
# and try to reestablish communication.
proc app_cmd {cmd} {
  global chan
  if {$chan != 0} {
    puts -nonewline "."; flush stdout
    set rc [catch {puts $chan "$cmd" ; flush $chan}]
    if {$rc != 0}{
        .top configure -bg "red"
        close $chan
        set chan 0
        after 1000 app_init}}}

# start widgets and render them to remote-host display
# frame is a container for the rest of the widgets
frame .top -bg red
pack .top

# create command buttons
button .top.forw -text "^" -command {app_cmd "forw"}
grid .top.forw -column 2 -row 1
button .top.back -text "v" -command {app_cmd "back"}
grid .top.back  -column 2 -row 3
button .top.righ -text ">>" -command {app_cmd "right"}
grid .top.righ -column 3 -row 2
button .top.left -text "<<" -command {app_cmd "left"}
grid .top.left -column 1 -row 2
button .top.stop -text "||" -command {app_cmd "stop"}
grid .top.stop -column 2 -row 2

# create status indications
# label .top.status -text "status" -bg "red"
# pack .top.status
button .exit -text "exit" -command {app_done}
pack .exit -side bottom

# start connection
# after calling app_init, interpreter enters even loop
app_init
```

to quickly write a program that has access to system resources, like files, timers, and most importantly here, the network interface.

Tcl has been ported to several OSs, including MacOS, Windows, Unixs, Linux, and DOS. It has a graphics toolkit-based widget library called Tk (see "Graphical User Interfaces in RTOSs," *INK* 94).

The client program tries to establish a network connection with the robot and implements a simple GUI using six buttons that let me send commands to the robot and exit. Photo 1a shows you what it looks like. The source code is given in Listing 2.

Notice that the screenshot in Photo 1a has a green background. The background switches to red, as seen in Photo 1b, when the client loses its network connection with the robot. When this happens, the client tries to reestablish the connection with the robot and turn the background green again.

The status and watchdog facilities are important, too, because these robots may run out of battery power or the RF may be too noisy because of interference or fading. It's convenient to have the GUI retry the connection, rather than having to restart it by hand.

REACHING OUT REAL TIME

Communication in real-time applications is simple today. Everyone used to implement their own protocols, which made sense when common communication libraries like TCP/IP weren't available or if you needed a mainframe computer to do it.

However, now that TCP/IP libraries are implemented with nearly every RTOS you're likely to find on 32-bit PCs, it makes sense to use it, even when you end up using a point-to-point, serial, or wireless link.

Although my robot uses a serial connection for its communication channel, that isn't the standard anymore for many applications. Ethernet is quickly becoming the standard communication interface on many SBCs and embedded peripherals. Most networked desktop PCs have an Ethernet port or it can be added inexpensively.

We're certain to see more Ethernet devices, and programming using the TCP/IP protocol will be standard. It maybe hidden, for example, when we use Web-based interfaces, but it's still there. RPC.EPC

Ingo Cyliax has been writing for INK *for two years on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego-based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. Before joining DSI, Ingo worked for over 12 years as a system and research engineer for several universities and as an independent consultant. You may reach him at cyliax@derivation.com.*

REFERENCES

D.E. Comer and D.L. Stevens, *Internetworking with TCP/IP: Volume III*, Prentice Hall, Upper Saddle River, NJ, 1997.

E. Foster-Johnson, *Graphical Applications with Tcl and Tk,* M&T Books, New York, NY, 1997.

J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

A. Tannenbaum, *Computer Networks*, Prentice-Hall, Upper Saddle River, NJ, 1996.

B.B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, Upper Saddle River, NJ, 1997.

SOURCES

Tcl/Tk Software
Scriptics Corp.
(650) 843-6900
Fax: (650) 843-6909
sunsoft.sun.com/tcl

Scriptics Corp.
(650) 843-6900
Fax: (650) 843-6909

Applied PCs

Fred Eady

A New View

Part 3: Sensors & Measurement Tools

In Fred's shop of virtual instruments, he installs a data-acquisition card and oscilloscope on his EXPL2, using VirtualBench. As a result, he can make just about any instrument he needs—and it's all in software.

I'll be the first to admit I'm wrong. May the i386EX gods and goddesses have mercy on my poor little 16-bit soul.

Last time, I scoffed at installing Bill's Windows 3.11 on my EXPLR2 because I figured it would be a waste of my paper and your time. I needed Windows 95 for my app. I'm big. I'm bad. I'm stupid.

It nagged me all month. Drove me nuts. That EXPLR2 board is perfect except for the processor type and speed. It has PC Card and everything else I need to put together a virtual instrument on that board. The i386 is the problem.

Wrong. I read so many datasheets that I tend to put everyday functions into datasheet form. So, I decided to scrap the datasheet search and install the Win95 hard drive I used for the National Instruments DAQ board on the EXPLR2.

It's been a while since I used the old EXPLR2, so I had to scrounge for some memory and haul out all the peripherals I used for the Tempustech/VIPer marriage. I plugged the EXPLR2 into the Vetra MegaSwitch and applied 60-Hz sine waves.

Well, look at that! Bill's logo is on the screen! This is gonna be fun!

I won't have to wade through the letters and E-mail telling me how "i386 stupid" I was today, so I can without conscience write this feature with extreme prejudice. (Remember that line from *Apocalypse Now*? I always thought it to be very military.)

Anyway, back to work. I've got a CD-ROM drive, 2.5-GB hard drive, mouse, and floppy on the EXPLR2. Only one problem exists. The floppy won't work.

I swap drives. I swap power connectors. I give up. It's got to be something I disabled when I did the first article with the board.

The last time I included EXPLR2 in a column, I used flash memory, and I really don't want to do the EXPLR2 doc search. I'm gonna fly by the seat of my Dockers this time!

Besides, I don't need that darned floppy drive anyway. I'm a CD kind of guy. I would normally use the floppy to move the screenshots to my Internet machine, but I can always tie the EXPLR2 to my Florida Room network and effect the transfer.

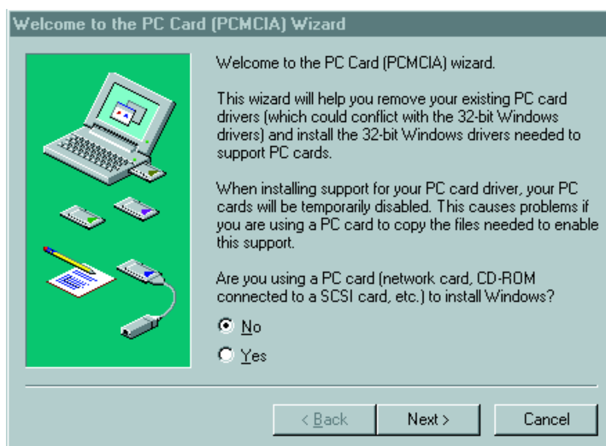


Photo 1—Even the almighty Merlin would not have been my choice at this point.

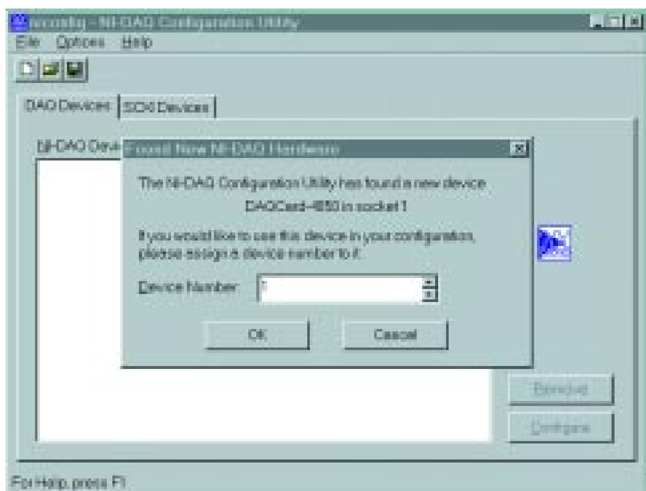


Photo 2—First the PC Card was found, and now the electronics have been identified. This is good.

Meanwhile, I found the EXPLR2 manual. I looked up floppy and hard disk installation. I read the manual. You know how drive A is always on the "twisted" end of the floppy cable? Yeah, that's OK for desktop monkeys, but the EXPLR2 likes the no-twist connector.

So, I plugged that floppy into the right connector—and boom! Floppy action. It was a good time to find the i386 datasheet, too. Things are rolling.

Hmm…. A cursory look at the features shows that the i386 is equipped with a full 32-bit internal architecture. (I've got to start reading my own stuff!) I really don't think the i386 was designed to do what I'm doing with it, but that's why I make the big bucks.

Photo 1 was like finding water in the middle of the dessert after the camel died. We're going to bring a National Instruments voltmeter and oscilloscope to life on the EXPLR, and the PC Card interface is the only way to go.

Using Bill's 3.11, I'd need to load Card and Socket Services. With Win95, I can forget about loading stuff I don't have. Besides, I don't have the ISA equivalents to the PC Card interfaces. They're at *INK* for photos. So, as my mom always says, "When in Rome, do spaghetti."

LET'S COOK

Last time, we put together a DAQ configuration that could acquire and generate digital and analog signals on command. This time, let's assemble the receiving end of that proposition. National Instruments also provides "instruments in a box," and I happen to have a couple of them here. We'll start with the voltmeter.

The documentation is a little flimsy, only 0.464″ including advertisements, but when a CD-ROM is included, you can bet things will get very particular once that little guy starts to spin. There's even a trio of diskettes in the package. I can use that floppy if I want to!

Nope, wait. There's another CD-ROM. Things looking a little complicated.

OK. Let's get a grip. This series is supposed to introduce you to National Instruments' way of doing things. So, with that in mind, I pick up the manual. Here we go.

THE VOLTS ARE IN

I love the way the National Instruments folks start their written dialog. "Thank you for buying a National Instruments DAQMeter DAQCard-4050. The DAQCard-4050 is a digital multimeter card for computers equipped with Type II PCMCIA slots." Couldn't have said it better myself.

The DAQCard-4050 I'm installing is a 5.5-digit DC voltage, true root mean square (RMS) AC voltage, ohm, and diode measurement device in PC Card form. In other words, this is a hand-held DMM in a PC Card.

Reading further, the DAQCard-4050 doc points out that the small size and low power consumption of the PC Card makes this little ditty ideal for portable computers in the field. Yep, and embedded computers in the field, too. A 24-bit ADC with digital filtering gives the DAQCard-4050 high resolution and accuracy while providing excellent noise rejection.

My application uses the VirtualBench-DMM software that's included with the package. VirtualBench is another name for a LabVIEW VI (virtual instrument) you would create except that it's already supplied by National Instruments in a canned, ready-to-roll format. You can still use LabVIEW and all other National Instrument programming packages to manipulate the DMM.

VirtualBench-DMM is preconfigured, and the intent of the prewritten VI is to make the DAQCard-4050 a voltmeter out of the box without you writing (or drawing, in this sense) any instrument code. You can, however, use C or even Visual Basic in conjunction with LabVIEW to program your own DMM instrument if you wish.

As you know, I tend to get a bit crabby about documentation when it's not quite up to par. Well, the folks at National Instruments must be reading my stuff because the documentation is very good.

The Read Me First doc is a flowchart that asks what National Instruments software platform or package you're using. The choices are VirtualBench-DMM, LabVIEW, or DAQCard-4050 Instrument Driver. Easy. I'm doing the VirtualBench thing.

The flowchart points me to the Getting Started with VirtualBench document. Well, the first thing it tells me is that I can't be here. This puppy requires a 33 MHz or higher '486 processor with floating-point capability. Hmm….

I may be in trouble here, but let's go after it anyway. After all, I'm still wearing my Dockers. But if this doesn't work for me, don't you try this at home.

Insert VirtualBench CD and select Run. Easy enough. It asked for the instrument I wished to install, and I selected the VirtualBench Suite. I'll install the scope hardware later.

As I was waiting for the software to install, I started thinking out my screenshots. I thought I'd better check out how high I could get the resolution to go on the EXPLR2.

It works out to 1024 × 768 with 16 colors or 800 × 600 with 256 colors. I ended up using the 800 × 600 setting.

Photo 3—And they said it couldn't be done.

The EXPLR2 doc states that a Cirrus Logic CL-GD6245 is present, but Bill's software showed a CL-GD6235 Rev. F. The IC imprint reads 6245. A quick Internet search for any special drivers I could install yielded nothing.

The VirtualBench software finally installed. I checked the directories and everything looked OK.

Installing the hardware was real tough. Inserting the PC Card and attaching the probe cable took all of 10 seconds. I powered up the EXPLR2 again with the new hardware and waited for smoke.

No smoke, but no DMM PC Card, either. The software informed me that I needed to run the NI-DAQ Configuration utility to install my hardware. Good thing I used the old LabVIEW drive, huh? The correct NI-DAQ software is included with the DMM. Using the LabVIEW drive just saved some time and effort.

Run NI-DAQ. Your wish is my command. Lo and behold, the NI-DAQ utility found my PC Card DMM. Photo 2 is legal documentation.

The detected base I/O address is 0x110–0x11F with an interrupt request setting of 10 in PC Card socket 1. There is a test button on this screen, and since I'm exploring with the EXPLR2, I clicked it. The test passed.

Smiling, I closed the NI-DAQ windows and restarted VirtualBench. Waiting. Waiting. Waiting…there—Photo 3!

Well, I'll show it to you as soon as I can fix my screen-capture software. Looks like I'll have to revert to the 16-bit version, as the 32-bit version is acting squirrelly.

HiJack for Win95 never works well for me. I end up loading the previous Win 3.11 version under 95. It worked better. Go figure.

Why don't you go to the fridge and get a drink and a sandwich while I load this? By the way, the EXPLR2 is equipped with 16 MB of memory and seems to be running OK for what it's loaded with. To be honest, I'm pretty amazed right now as to how well it is running.

The voltmeter refresh is slow but usable. After all, this really isn't supposed to be working at all.

I couldn't resist. I probed the +5-V line on the power supply feeding the EXPLR2. Photo 4a popped up after about 5–10 s. Pretty close, I'd say.

From the looks of Photo 3 and 4a , there are lots of things you can do with the

Photo 4a—Yes! It's pretty accurate for hardware that's not supposed to be running this app. b—This DMM is really tricky.

canned version of the DMM. The front panel is broken down into areas. The range-selector area consists of the typical voltage range settings you find on your hand-held meter.

The measurement display area is obvious, as is the mode-selection area. The main control area resides under the display are and consists of a cluster of math buttons and the run and log buttons.

One of the more interesting math buttons is the max/min button depicted by a graphic of a full and not-so-full vessel. When pressed, a minimum and maximum value of voltage input is displayed above the current voltage reading, as you see in Photo 4b.

I did a little digging and found that the DMM was set for 50-Hz operation. I changed that and took some readings. Didn't seem to change anything.

The documentation says that the Power Line Frequency control reduces measurement inaccuracies caused by line noise and interference from AC-powered equipment and overhead lighting. Also, the digits of precision can be changed to increase or decrease accuracy. The more digits, the higher the accuracy and the slower the measurement time.

The math buttons enable you to scale the readings in almost any way you like. Basically, you input a multiplier value and an offset value, and the DMM calculates the output value accordingly. The decibel measurement can be altered by manipulating the percent of reference and the impedance. The data-logging option lets you take a reading at a specific interval and store it in a file.

I recall writing a battery-analysis application that took months to complete. In less than two hours, I loaded the DMM, and it can do more battery analysis than what I wrote in all those months.

I could go into the datasheet specifics of the virtual DMM, but let me sum it all up by telling you the virtual DMM specs look like your hand-held specs.

SCOPE IT OUT

The introduction to the DAQScope 5102 thanks you for your participation and goes

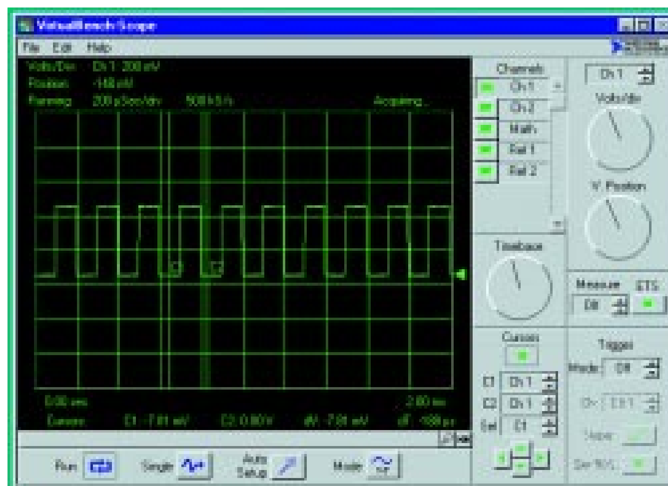


Photo 5—Notice the cursors. This thing looks just like my big ol' Tek, which by the way is supplying the waveform from its calibration output.

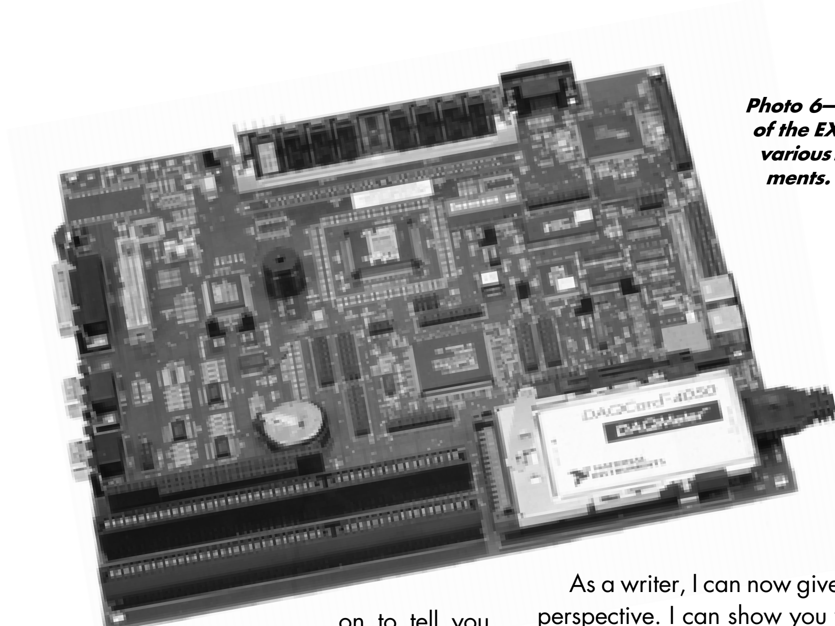


Photo 6—Here's a shot of the EXPLR2 with the various PC Card instruments.

on to tell you what a magnificent instrument you just purchased. The DAQScope 5102 features two 8-bit resolution analog-input channels with a real-time sampling rate of 20 MS/s to 1 kS/s.

The input bandwidth is 15 MHz with multiple trigger options. If you need more than two channels, you can synchronize multiple devices using RTSI (real-time system integration) bus triggers or PFI digital triggers on the I/O connector.

I installed the DAQScope PC Card and did the smoke test. No smoke and, again, no PC Card. NI-DAQ to the rescue. The DAQScope weighed in at base address 0x120–0x13F on interrupt 10 in socket 1.

The documentation guides you through the hardware and software installation and then tutors you on digitizing basics. They just want you to know the environment you're measuring in.

Photo 5 is the justification. I could go on extolling the virtues of the DAQScope, but like the DMM, it behaves much like the monster on your bench. Is this neat or what? The main hardware sans cables and common peripherals is shown in Photo 6.

WHERE ARE WE GOING?

As an engineer, I'm thrilled with the possibilities the National Instruments package offers. I can now put just about any instrument I can imagine on an embedded platform. I'm no longer choked by size and power consumption. The software is the instrument.

The way my virtual instruments behave depends entirely on the platform I choose. Here, I put together a system using a combination of components you'd probably never use in a real-world situation. The point is that it worked and it wasn't supposed to.

As a writer, I can now give you a better perspective. I can show you waveforms. I can show you the front panels of the instruments in relation to the waveforms that instrument may acquire or generate. I can assemble a virtual-instrument suite and show you every aspect of each particular instrument of interest. As you've probably guessed, I'll continue to use these virtual instruments in future articles.

This time around, we combined the resources of an old friend, the EXPLR2, with a new friend, LabVIEW, to prove once again it doesn't have to be complicated to be embedded. APC.EPC

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

SOURCES

EXPLR2
RadiSys Corp.
(800) 950-0044
(503) 646-1800
Fax: (503) 646-1850
www.radisys.com

LabVIEW
National Instruments Corp.
(512) 794-0100
Fax: (512) 794-8411
www.natinst.com

VMAX SBC 301
Tempustech
(941) 643-2424
Fax: (941) 643-4981
www.tempustech.com

VIPer806
Teknor Microsystems, Inc.
(800) 387-4222
(561) 883-6191
Fax: (561) 883-6690
www.teknor.com

MegaSwitch
Vetra Systems Corp.
(516) 434-3185
Fax: (516) 434-3516
www.vetra.com

Using a PC for Radiation Detection

FEATURE ARTICLE

Dan Cross-Cole

Modifications for Multichannel Analyzer Capability

Having a problem with radon? Using a ADC board and his desktop computer, Dan can measure radon levels as well as the energy of the incoming radiation. The program even shows him what to do to keep radon levels within EPA limits.

Of all the radiation detectors available for home computers, you'll find the cheapest are Geiger-tube types costing under $200. But, these devices give limited information about what's causing the radiation because they provide only a constant amplitude output pulse without any data on the energy of the incoming radiation.

A typical Geiger tube consists of a metal cylinder at one voltage (usually near ground) and a thin wire along the axis inside the cylinder at a high voltage (e.g., 700 V). In the tube, a mixture of gases causes an avalanche of electrons when a pulse of electrons enters the cylinder. That is, one electron hits a gas molecule and knocks off several more electrons, each hitting another gas molecule and knocking off more electrons.

As a result, the tube appears as a low resistance for an instant (~100 µs). A high-value resistor in the power supply causes the voltage across the tube to decrease, which, combined with the effects of the gas mixture, halts the avalanche.

The pulse of electrons starting all this comes from an interaction between the metal cylinder and a gamma ray. The gamma ray acts like a high-speed billiard ball, knocking electrons into the gas inside the cylinder.

The Geiger tube is usually coupled to a counter via a capacitor that passes the pulse generated by the Geiger tube to the counting electronics. Geiger tubes can furnish pulses that directly drive small LCD counters available from Radio Shack.

A standard setup consists of a Geiger tube, power supply, and pulse-counting circuit. This circuit only tells you how many gamma rays were detected. It doesn't give you any information about the energy of the gamma rays.

Gamma rays originating in the nucleus of the atom are emitted with known energies. So, you can use the patterns to determine what types of atoms are the source of the radiation.

Radium, for example, has a well-known pattern of energies given off at 609, 352, 295, and 242 kilo-electron volts (keV). (Actually, the energy is given off by the lead in the decay process.) Similarly, cesium has characteristic emissions at 184 and 662 keV.

The detector for a gamma spectrometer provides an output pulse proportional to the energy of the incoming gamma ray. You can expect to pay about $800 for the detector and $200 for the power supply. The result is a

Photo 1—*On this modified Prairie Digital Model 30 ADC board, the white wire is the clear (charge dump) signal and the shielded wire is for the pulse input. Note the two 1/8″ phone jacks.*

gamma spectrum (see Figure 1), which you can use with standard handbooks to identify the source of radiation (e.g., radon).

A typical gamma spectrometer consists of an NaI crystal coupled to a photomultiplier tube. Gamma rays entering the crystal knock a pulse of electrons into higher energy states in the crystal. When the electrons return to their lower states, light photons are given off.

The number of photons emitted is proportional to the energy of the incoming gamma ray. The photomultiplier tube amplifies this photon pulse to give a voltage pulse, which is amplified and sent to the ADC. The ADC has a range of 0–5 V.

Remember, the peak voltage of the output pulse is proportional to the energy of the incoming gamma ray. If you vary the amplifier gain so the peak of the cesium energy peak (662 keV) corresponds to 3.31 V, then a pulse at 1 V is equivalent to a gamma ray with an energy of 200 keV.

The added cost for the ability to identify radiation sources is out of reach for almost everyone except professional labs. But here, I show you how to modify an inexpensive (under $100) ADC board for your computer. Using Visual Basic for Windows, you can obtain the functions of a multichannel analyzer at a more reasonable cost.

Of course, you're still stuck with the high price of the detector. But, reduced funding for the supercollider may result in an abundance of these detectors hitting the surplus market.

The program operates in both Windows 3.1 and 95. But, the specs on the Prairie Digital board limit the sampling rate to 40,000 samples per second.

On a 20-MHz '386, the program performs about 6000 samples per second. Source code for the C++ sampling loop is also provided. Processors with higher speeds may require delay loops.

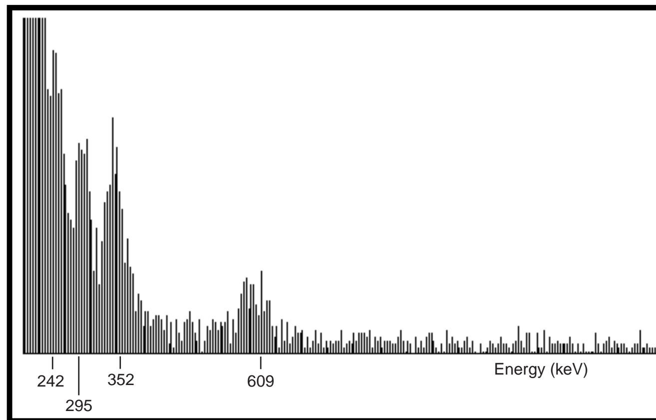


Figure 1—*The vertical scale is 100 counts in this channel-versus-energy graph. This is a typical spectrum for radium. The spectral lines shown are from the daughter products of radon. The ADC board has eight-bit resolution, giving 256 channels.*

MODIFY THE BOARD

To the ADC board (see Photo 1), I added two ⅛″ phone jacks for the voltage input from the sample-and-hold circuit as well as the charge dump signal to the same circuit. You can also see the position of the jumper pins.

The phone jacks are secured to the board with two thin sheets of aluminum—one on each side of the steel bracket that comes with the board. In its literature, Prairie Digital states that it can provide customized inputs.

The pulse-clear signal comes from pin B9 of the user connector. This single wire also attaches to one of the ⅛″ phone jacks (Clear). The voltage input connects via a shielded cable to pin A24. Pin A0 is for grounding the shield of the cable. The shield is also connected to the ground of the other ⅛″ phone jack (Vin).

PUT IT IN THE SOFTWARE

The key to this Visual Basic application is that events happen when you click an object onscreen with a mouse. Photo 2 shows the control screen for the gamma spectrometer.

You enter the vertical scale of the graph and the number of samples to collect in the input boxes. Auto Sample causes the computer to take 65,500 data points for each sample and display them onscreen.

Push buttons let you add the data to previous samples or display each sample separately. You can take a single sample, stop and check the clipboard to see that the hardware is working, and then graph the single-sample data. From the control screen, you can store the data in a text file or retrieve a file for display.

The device has storage for two channels—P1 and P2. This storage is used both for energy calibration and for summing the counts under an energy peak.

For energy calibration, constants are entered for two peaks of known energy value in the blocks labeled `P1` and `P2` under `Calibration Points (keV)`. For example, a radium dial watch has two well-known peaks at 352 and 609 keV.

When I place the mouse on the channel corresponding to the lower energy peak (P1) in the graph display area, the channel number appears in the `CHANNEL #` block and then I have to `Store P1`. When `Calibrate keV` is clicked, the energy corresponding to any channel selected by the mouse is displayed in the `keV` block.

P1 and P2 can also obtain the signal counts under an energy peak. In this case, P1 and P2 not only contain the channel number but also the counts per channel corresponding to the vertical coordinate of the mouse cursor in the graph.

In this way, the cursor can be placed at the noise level of the graph, so only the signal counts are summed by the program. After I select P1 and P2, `Find Counts in Peak` causes the total number of counts under the selected peak to appear.

Listing 1 shows code for `Command5`, the `Auto Sample` button. All of the Visual Basic code operates on a single string of data taken from the clipboard.

The Visual Basic program calls the C++ routine (`Portdata`), which oper-

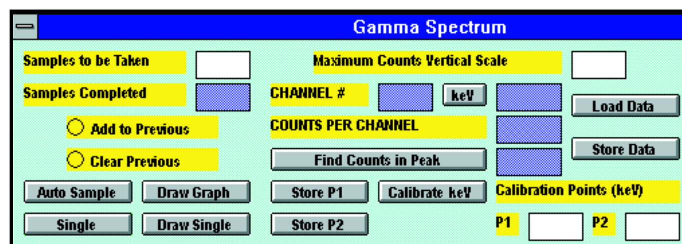


Photo 2—*Clicking the mouse while the cursor is on the controls causes the system to sample and display data on this control panel for the gamma spectrometer.*

ates the hardware ports, collects the samples, and stores the data in the clipboard. Visual Basic then displays that data on the gamma spectrum.

Listing 2 shows the source code for `Portdata`, which I wrote in Borland's Turbo C++ for Windows. It is furnished to the Visual Basic program as a dynamic linked library (`DASADC.DLL`).

If you're thinking of taking on this project, there are two things to note: You don't need to know a lot of Windows programming. But, you do need to know some C++ syntax.

The C++ port commands (`inportb`, `outportb`) receive or send individual bytes of data to the ADC board. `Portdata` collects 65,500 data points and, depending on the digitized pulse amplitude, increments one of 256 bins.

After each data point is digitized, a positive trigger signal is applied to the sample-and-hold circuit to dump the charge and prepare for the next pulse. After all data points are taken, the data is converted to a string of 256 numbers and sent to the clipboard.

Using a 20-MHz '386, this system takes data at a rate of about 6000 data points per second. This speed is about 100 times faster than the detector pulse rate at environmental radiation levels.

Listing 1—*In this Visual Basic subroutine,* `Command5` *is activated when the* `Auto Sample` *button on the control panel is clicked with the mouse. The subroutine then takes the number of samples that were entered into the* `Samples to be taken` *box.*

```
Sub Command5_Click ()         'Auto Sample is clicked
  RunNum = Val(Text1.Text)    'Samples to be taken
  For J = 1 To RunNum
    X = 1
    Call PORTDATA(X)          'Call C++ procedure
    Command3.SetFocus         'Call Draw Graph subroutine
    Command3.Value = -1
    Text2.Text = Str$(J)      'Write to Samples Completed Box on
                                 control panel
  Next J
End Sub
```

WHAT GOOD IS IT?

This system gives a gamma spectrum from 100 keV to 1.1 MeV, but the range can be adjusted by varying the high voltage and the pulse amplifier gain. The present range detects common gamma emitters and is especially suited to monitor the daughter products of radon.

The present detector (a 1″ diameter, 1″ long crystal) is suitable for monitoring an area to see if levels significantly exceed environmental guidelines. Measurements from this system helped me determine that all I needed was a small muffin fan to keep my basement below EPA radon levels.

The easiest way to calibrate the system for gamma energy is to find an old radium watch—the kind that glows in the dark without being put under a bright light. One that is painted green but doesn't glow anymore may still provide a good calibration signal.

The easiest way to adjust gain (i.e., energy range) is to change the feedback resistor of the pulse amplifier. Higher resistance yields higher gain.

The system can be calibrated for activity if a source of known intensity is available. Schools and laboratories may have these types of calibration sources.

Using a separate hardware program has one good design advantage. You can change the hardware and a relatively simple C++ program without changing the Visual Basic application.

For example, the present C++ routine clocks the data out of the ADC chip. A faster design is to have an onboard clock on the ADC board, which would provide 30,000 data points per second. The C++ routine is simpler because the hardware clocks the data.

MEASURING RADON SOURCES

First, let's calibrate the system. My detector is wrapped with a lead sheet approximately 3/32″ thick to form a tube about 2.5″ in diameter, with the detector crystal photomultiplier tube tucked inside. The wrapping extends about 2.5″ past the end of the detector crystal, so that's where I place the check source (5.0 µCi of Cesium-137).

The detector measures the activity of my basement floor (~10″ of concrete). The detector face looks straight into the floor (~ 2.5″ above it). A lead sheet reduces stray radiation from the side.

The Cesium button gives 2631 counts under the 662-keV peak. The activity of the button is 5.053 µCi with 50 samples (of 65,500 data points each).

After taking 5000 samples, I measured 134 counts under the U-235 peak at 185 keV. Radiation handbooks say 54% of the U-235 is changed by emitting the 185-keV gamma ray.

So, the activity of U-235 is:

$$134\text{ counts}\left(\frac{5.05\ \mu\text{Ci}}{2631\text{ counts}}\right)\left(\frac{1}{0.54}\right)\left(\frac{50}{5000}\right) = 0.00476\ \mu\text{Ci}$$

I can calculate the number of U-235 atoms that give rise to this activity. Given the diameter of my lead shield (2.5″), the activity per area is:

$$\frac{0.00476\ \mu\text{Ci}}{\left(3.1416\,\frac{(2.5\times 2.5)}{4}\right)} = 1\text{ nCi / in.}^2$$
$$= 0.37\text{ Be / in.}^2$$
$$= 0.37\text{ disintegrations/ s / in.}^2$$

Listing 2—*This C++ program for* `portdata` *operates the hardware ports that control the ADC board.*

```
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <dos.h>

  static HINSTANCE hInstance;     //Windows variables and constants
  static HINSTANCE hPrevInstance;
  static HINSTANCE hGlobalMemory;
  const HWND near hClip = 0;
  static LPSTR lpGlobalMemory;
```

(continued)

Listing 2—*continued*

```
  static LPSTR lpSzBuffer;
  static int nCmdShow;
  static int MessageLoop(void);
  static short nLength;
int FAR PASCAL LibMain(HINSTANCE hInstance, WORD wDataSeg, WORD
                       wHeapSize, LPSTR lpszCmdLine){
  if (wHeapSize > 0) UnlockData (0) ;
  return 1;}
//Routine is inserted into standard DLL format
extern "C" void FAR PASCAL _export portdata(int){
  int portc = 774;                /these are specific to the
  int portd = 775;                /Prairie Digital ADC board
  unsigned char reset = 129;
  unsigned char clkhi = 64;
  unsigned char clklo = 0;
  unsigned char adcstart = 192;
  unsigned char bitget = 8;
  unsigned char adcdata;
  unsigned char adcbin;
  unsigned char adcbina;
  unsigned char bitbin;
  unsigned char bitbina;
  unsigned char bitbinb;
  char SzBuffer[5200];            /buffer for data
  unsigned int SampCnt;
  unsigned int BinNum;
  unsigned int TimeCnt;
  unsigned int n = 0;
  unsigned int SpecData[257];
  char TmpBuffer[20];
  unsigned Sample = 65500;
  for (BinNum = 0; BinNum <= 256; BinNum++){
    SpecData[BinNum] = 0;}
  asm cli
  outportb(portd,reset);
  outportb(portc,adcstart);
  for (SampCnt = 1; SampCnt <= Sample; SampCnt++){
    adcbin = 0;
    adcbina = 0;
    outportb(portc,clkhi);
    for (BinNum = 0; BinNum <= 7; BinNum++){
      adcdata = inportb(portc);   /get data from ADC board
      bitbin = adcdata & bitget;
      bitbina = bitbin << 4;
      bitbinb = bitbina >> BinNum;
      adcbin = adcbina | bitbinb;
      adcbina = adcbin;
      outportb(portc,clklo);
      outportb(portc,clkhi);}
    outportb(portc,adcstart);
    ++(SpecData[adcbin]);}
  asm sti
  for (BinNum = 0; BinNum <= 256; BinNum++){
    itoa(SpecData[BinNum],TmpBuffer,10);
    lpSzBuffer = lstrcat (SzBuffer, TmpBuffer); /add data to buffer
    lpSzBuffer = lstrcat (SzBuffer, "\n");}  /add return, line feed
  hGlobalMemory = GlobalAlloc(GHND,(DWORD) 5200);
  (void far *)lpGlobalMemory = GlobalLock(hGlobalMemory);
  for (n = 0; n < 5200; n++){
    *lpGlobalMemory++ = *lpSzBuffer++;}
  GlobalUnlock(hGlobalMemory);
  OpenClipboard(hClip);
  EmptyClipboard();
  SetClipboardData(CF_TEXT,hGlobalMemory);   /put data in clipboard
  CloseClipboard();
  *SzBuffer =NULL; }
```

I relate this to the number of atoms through the radiation-decay formula:

$$N = N_0 e^{-0.693 t / T} \qquad [1]$$

where T is the half-life of the element. I then differentiate this formula to obtain:

$$\frac{dN}{dt} = \left(\frac{-0.693}{T}\right) N$$

At dN/dt is 0.37 disintegrations per second, I get:

$$N = 0.37 \left(\frac{T}{0.693}\right) \text{atoms}$$

where T is the half-life of U-235 in seconds:

$$\left(7.1 \times 100{,}000{,}000 \text{ yr.}\right)\left(\frac{365 \text{ day}}{\text{yr.}}\right)\left(\frac{24 \text{ h}}{\text{day}}\right)\left(\frac{3600 \text{ s}}{\text{h}}\right)$$
$$= 2.239 \times 10^{16} \text{ s}$$

The number of atoms N seen through the 2.5″ diameter disc of concrete floor is 1.2×10^{16}.

The range of gammas detected extends several inches into the concrete, but I want to get the atoms near the surface that could emit radon. Atoms further into the concrete stay embedded as they change.

Let's consider the range of alpha particles in concrete. Most alphas are stopped by about 0.03 mm of concrete. Radon is stopped more easily because it's larger and slower. I'm being conservative by using the alpha-particle range.

For now let's consider the concrete as a series of discs 2.5″ in diameter and 0.03 mm thick. Eventually, I'll use the surface of the basement walls and floor.

Each disc has the same activity, but the detector sees less of it because of the shielding effect of the discs. Essentially, we're looking at an effective volume that gives the total detector reading.

Determining this effective volume gives a density of atoms. From that, I derive the density of atoms on the surface of the concrete, which are the ones that can emit radon.

To get the effective volume, I looked up the half-value layer for concrete and used the formula for shielding, which is similar to that for radiation decay:

$$\text{Intensity} = I_0 e^{-0.693 x / X}$$

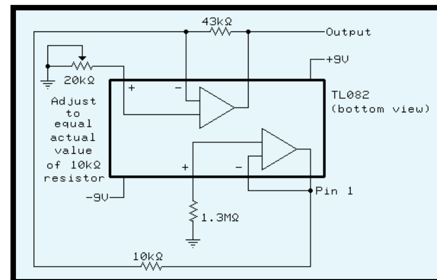


Figure 2— *I used 9-V batteries for my power supply in the detector's pulse amplifier circuit. This circuit converts the millivolt-sized pulses from the photomultiplier tube of the detector to volt-sized pulses for the ADC board.*

where X is the half-value layer or the thickness of material needed to lower the intensity by half.

For concrete at 200 keV, the half-value layer is about 1″. By integrating the exponential between 0″ and 10″, I found that the effective depth is 1.44″.

If A_s equals the surface activity counted by the detector and A_v is the activity of the effective volume, then:

$$A_s \left(3.1416 \times \left(\frac{2.5}{2}\right)^2\right)$$
$$= A_v \left(3.1416 \times \left(\frac{2.5}{2}\right)^2\right) \times 1.44$$

In other words, the activity is the same whether I treat it as a surface (disintegrations per in.^{2}) or volume (disintegrations per in.^{3}). I measured surface activity and can calculate volume activity using this formula:

$$A_v = \frac{A_s}{1.44}$$

Since the surface activity is:

$$\begin{aligned} A &= 1 \text{ nCi / in.}^2 \\ &= 0.37 \text{ disintegrations/ s / in.}^2 \end{aligned}$$

the volume activity A_v is:

$$\begin{aligned} \frac{0.37}{1.44} &= 0.257 \text{ disintegrations/ s / in.}^3 \\ &= 2.7 \times 10^{-10} \text{ Ci / in.}^3 \end{aligned}$$

Let's focus on the thin disc of surface concrete (2.5″ diameter, 0.03 mm thick) seen by the detector. From A_v, the total activity of the disc equals:

$$\begin{aligned} A_v\left(3.1416 \times \left(\frac{2.5}{2}\right)^2\right)\left(0.03 \times \frac{1}{25.4}\right) &= 0.257 \times 9.87 \times 0.00118 \\ &= 0.003 \text{ disintegrations/ s} \end{aligned}$$

Recall equation 1 which relates activity to the total number of radioactive atoms. Using 0.003 for dN/dt and 2.239×10^{16} seconds for T (the half-life of U-235), I obtain:

$$\begin{aligned} N &= \left(\frac{0.003\left(2.239 \times 10^{16}\right)}{0.693}\right) \\ &= 9.69 \times 10^{13} \text{ U–235 atoms} \end{aligned}$$

Now, I can find the number of U-238 atoms.

Fortunately, the ratio of U-235 to U-238 atoms is known. In natural Uranium, the abundance of U-235 is 0.7196%. The rest is mostly U-238.

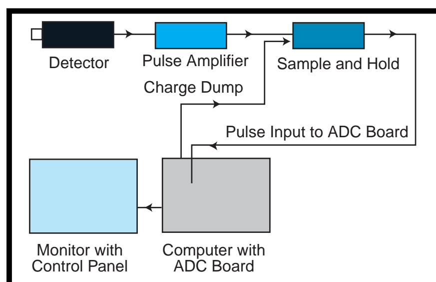


Figure 4—*The ADC board and Visual Basic function as a multichannel analyzer. The ADC board plugs into the computer's ISA bus. The detector, amplifier, and sample-and-hold circuit are external to the computer. The ADC board is modified to contain two ⅛″ phone jacks for the pulse input and the charge dump signal.*

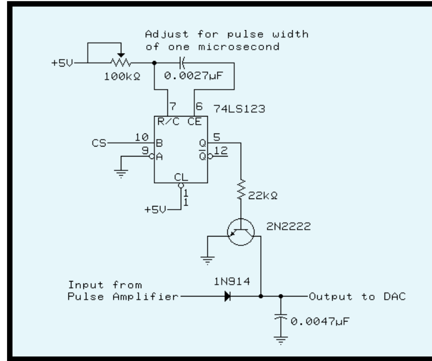


Figure 3—*In this sample-and-hold circuit, the transistor is normally off while the circuit is waiting for a pulse from the pulse amplifier. The diode keeps the peak voltage of the pulse on the capacitor long enough for the ADC to sample the voltage and send a clear signal to the CS input of the 74LS123, which pulses the transistor, dumping the charge to ground.*

I want to calculate U-238 because radon comes from the decay of U-238. Given the amount of U-235, the amount of U-238 is:

$$\begin{aligned} N_{238} &= \frac{9.69 \times 10^{13}}{.007196} \\ &= 1.35 \times 10^{16} \text{ atoms} \end{aligned}$$

The activity of these atoms can be calculated, again using equation 1. In this case, T equals the half-life of U-238 (1.42×10^{17} s):

$$\begin{aligned} \frac{dN}{dt} &= \left(\frac{-0.693}{1.42 \times 10^{17}}\right)\left(1.35 \times 10^{16}\right) \\ &= -6.59 \times 10^{-2} \text{ atoms / s} \end{aligned}$$

Dividing this total U-238 activity in the disc by the area of the disc gives me the rate of radon production:

$$\begin{aligned} \frac{dN}{dt} / \text{in.}^2 &= \left(\frac{-6.59 \times 10^{-2}}{3.1416 \times \left(\frac{2.5}{2}\right)^2}\right) \\ &= -1.34 \times 10^{-2} \text{ atoms / in.}^2 \end{aligned}$$

Since each U-238 atom becomes a radon atom, this is the rate of production of radon per square inch of concrete surface.

In my basement, two walls measure 20′ × 8′, two walls 24′ × 8′, and a floor of 24′ × 20′, which makes a total of 170,496 in.^{2}. In 1 s, the total number of radon atoms produced at the surface is:

$$1.34 \times 10^{-2} \times 170{,}496 = 2.285 \times 10^{3} \text{ atoms}$$

Using equation 1, let's compare this with the EPA recommended limit for radon (4×10^{-12} Ci/l). Here, T is the half-life of radon, 3.82 days or 330,048 s:

$$\begin{aligned} \frac{dN}{dt} &= \left(\frac{-0.693}{330{,}048}\right)\left(2.285 \times 10^{3}\right) \\ &= -4.8 \times 10^{-3} \text{ atoms / s} \\ &= \left(-4.8 \times 10^{-3}\right)\left(\frac{1 \text{ Ci}}{3.7 \times 10^{10} \text{ Beq}}\right) \\ &= 1.3 \times 10^{-13} \text{ Ci / s} \end{aligned}$$

Given the typical basement's volume of 1.09×10^5 l, how long does it take to reach the EPA limit? The total activity for the EPA limit is:

$$4 \times 10^{-12} \text{ Ci / l} \times 1.09 \times 10^{5} \text{ l} = 4.36 \times 10^{-7} \text{ Ci}$$

and the time t to reach this activity is:

$$\begin{aligned} t &= \frac{4 \times 10^{-7} \text{ Ci}}{1.3 \times 10^{-13} \text{ Ci / s}} \\ &= 3.35 \times 10^{6} \text{ s} \end{aligned}$$

or in a more useful unit:

$$\begin{aligned} 3.35 \times 10^{6} \text{ s} &= \left(3.35 \times 10^{6}\right)\left(\frac{1 \text{ h}}{3600}\right)\left(\frac{1 \text{ day}}{24 \text{ h}}\right) \\ &= 38.8 \text{ days} \end{aligned}$$

To clear your basement every 40 days, you need a fan that exhausts at least 1.09×10^5 l in that time. To convert to units normally used for fans (ft.^{3}/min.), first note that the typical basement (24′ × 20′ × 8′) is 3840 ft.^{3} The minimum capacity, then, is 3840 ft.^{3} in 55,872 (38.8 × 24 × 60) min.:

$$\frac{3840}{55{,}872} = 0.069 \text{ ft.}^3 \text{ / min.}$$

Typical muffin fans are about 30 ft.^{3} per minute in airflow. You can evacuate the 3840-ft.^{3} volume in 128 min., which is just a little over 2 h. Since half the radon decays in 3.82 days, you certainly stay below the EPA limit by running the fan for a couple hours each month.

You see, if you have a poured concrete basement, you may not need an expensive solution to radon buildup.

ADDITIONAL HARDWARE

A pulse amplifier raises the millivolt-sized pulses from the detector to pulses with amplitudes between 0 and 5 VDC. A suggested pulse-amplifier circuit is shown in Figure 2.

It should be assembled in a shielded enclosure, with no more than 1′ of shielded cable between the detector output and the amplifier input. This

amplifier also inverts the polarity of the negative detector pulses to the positive pulses required by the ADC board.

A sample-and-hold circuit captures the pulses and holds them for the ADC board. The clear (charge dump) signal from the modified ADC board grounds the pulse voltage after its peak is measured (see Figure 3).

Figure 4 shows the entire setup. To reduce background radiation, the detector was surrounded with a 3⁄32″ lead sheet.

Standard shielded audio cable connects the sample-and-hold circuit to the ADC board and the pulse amplifier. The length here isn't critical—up to 10′ is fine. The cable between the detector and the pulse amplifier input shouldn't be longer than 1′ and should be RG-58–type cable or similar.

CIRCUIT USES

In addition to serving as a gamma spectrometer, the circuit can act as a multichannel analyzer to measure the height of pulses between 0 and 5 V. It can detect irregularities in pulse heights from a signal generator. The ADC board digitizes audio waveforms without the sample-and-hold circuit.

The sample-and-hold circuit is biased toward larger pulses. It records the highest pulse received between charge dumps. This is not a problem for background radiation with its relatively few pulses per second. But for audio, the distortion would be unacceptable.

If you're serious about measuring radon, however, you should use several methods of measurement. One highly recommended option is to use a carbon canister that is then sent to a laboratory for testing. You use a piece of filter paper with a known quantity of air passing through it, measure for radon daughter products, and determine the air concentration of radon.

Or, you can just install a fan. ◼

The views expressed in this article do not necessarily reflect the views of the Department of Navy or the United States government.

Dan Cross-Cole has worked as an electronics engineer for the Navy for almost 18 years. His interest in instrumentation for personal computers dates back to the Altair 680b (circa 1976), using an A/D conversion routine and hardware to display the recorded data on an oscilloscope. Dan is also the author of 26 Hardware Projects for Your Home Computer. *You may reach him at crosscol@erols.com.*

SOURCES

NaI detector assembly, voltage divider
Bicron Corp.
(216) 564-2251
Fax: (216) 564-8047
www.bicron.com

HVPS MOD 1-kV POS OPT 3
Bertan Associates
(516) 433-3110
Fax: (516) 935-1766
www.bertan.com

Model 30 general-purpose data-acquisition board
Prairie Digital, Inc.
(608) 643-8599
Fax: (608) 643-6754
prairdig@bankpds.com
www.prairiedigital.com

DEPARTMENTS

66 MicroSeries

74 From the Bench

80 Silicon Update

FreeDOS and the Embedded Developer

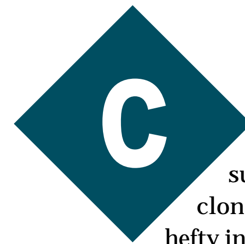
Using the Kernel

MICRO SERIES

Pat Villani

Part 2 of 2

Last month, Pat gave us the rundown on FreeDOS, a royalty-free operating system for embedded systems. This month, he shows us how to make it fly with an intelligent printing application for airline tickets and boarding passes.



ommercial OSs such as MS-DOS clones come with a hefty initial license fee and per-copy royalties that buoy the cost of your final product. These royalties often range from a few dollars to tens of dollars per copy. In a competitive market, these costs are significant.

Last month, however, I introduced you to FreeDOS. Under the terms of the GNU public license, you can write an application and take advantage of a royalty-free environment, which lowers recurring costs on the final product.

I discussed how the FreeDOS kernel becomes compatible by localizing compatibility code to the interface layers. I showed how to load the kernel into memory.

The kernel is a standard DOS `.EXE` file, which eliminates special relocation code usually found in other DOS systems. FreeDOS and DOS-C accomplish this via an intermediate program loader that is aware of the `.EXE` file format and that performs all the necessary relocation.

Using an OS like FreeDOS simplifies your design by handling the device

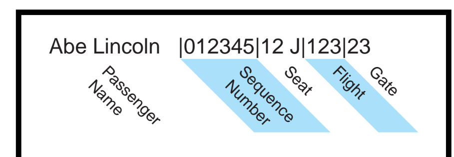
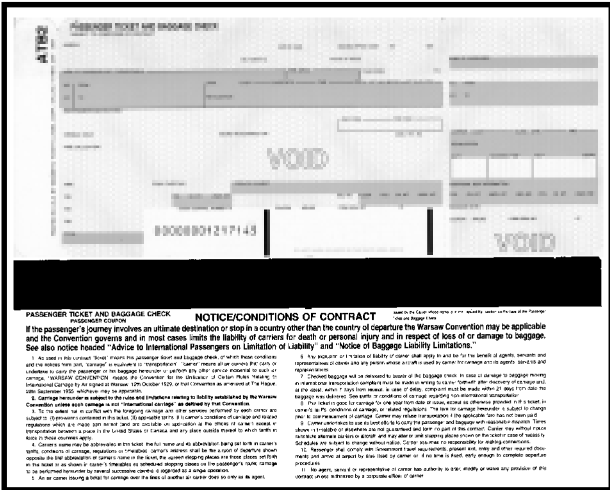


Figure 1—*The passenger record for the boarding-pass software includes the passenger's name and sequence number, as well as seat, flight, and gate numbers.*



PASSENGER TICKET AND BAGGAGE CHECK
PASSENGER COUPON

NOTICE/CONDITIONS OF CONTRACT

If the passenger's journey involves an ultimate destination or stop in a country other than the country of departure the Warsaw Convention may be applicable and the Convention governs and in most cases limits the liability of carriers for death or personal injury and in respect of loss of or damage to baggage. See also notice headed "Advice to International Passengers on Limitation of Liability" and "Notice of Baggage Liability Limitations."

2. Carriage hereunder is subject to the rules and limitations relating to liability established by the Warsaw Convention unless such carriage is not "international carriage" as defined by that Convention.

Photo 1—*This sample ATB document shows all the preprinted fields. Note the magnetic stripe on the reverse side.*

drivers and offering a standard API to the embedded application. It's quite similar to a typical DOS system, except it's embedded.

This month, I want to take this knowledge and apply it. An intelligent ticket printer provides a great opportunity to see the possibilities for using FreeDOS and DOS-C in an embedded system.

ATB PRINTERS

At one time, airline tickets were multipart tickets requiring a typewriter or impact printer to create an image on all parts of the form. The airlines manually tracked passengers and computed revenue by counting the tickets collected. This slow, manual procedure was labor intensive and costly.

As computers became part of their daily operations, air fares lowered and more flights were scheduled. But, more flights means more passengers. To continue to offer low-cost fares, the airlines needed to improve passenger movement through airports or they'd lose revenue because unhappy customers would find other ways to travel.

Improving passenger movement required a change in the ticketing procedure. It needed to be automated with tickets that could be produced quickly and efficiently. Also, passengers had to be able to board quickly and efficiently.

These factors led to the development of intelligent ticketing printers. The printers produce tickets and boarding passes on cardboard-like stock of Photo 1. Information is encoded on a magnetic stripe on the tickets' reverse side.

Now, airlines can use magnetic readers in their ticket accounting systems, which speeds up ticket handling and reduces costs.

Unlike the printer attached to your development system, these printers handle data in an encoded record where each field in the record contains information regarding placement.

The record may also contain embedded control information. One airline even created an interpreted language like Pascal p-code or Java VM byte code to customize record handling. These schemes shift the workload of formatting the documents to the printers, free CPU time to handle other tasks, and reduce cost through lowered overhead.

HARDWARE ARCHITECTURE

The hardware required to generate this type of document can be quite complex. For example, a full ATB document requires two distinct processes—one to encode the magnetic stripe on the reverse of the document and another to print the information on the obverse of the document.

Discussing such a printer is beyond the scope of this article, nor am I going to cover the theory behind the paper handling. I do want to discuss a representative printer that prints a simpler boarding-pass document. It takes a record like Figure 1 and creates a document like the ones shown in Photo 2.

A good place to start is to conceptually break apart the functionality required to build an intelligent printer. As you see in Figure 2, there is, of course, the print mechanism and the surrounding electronics. This so-called print engine is the most fundamental part of the printer.

Also, the communications interface connects the printer with a host. And in between, there is the processing required to interpret the data records and image the documents.

In many traditional designs, all this functionality is rolled up into a single, custom-designed board. This board is usually large and expensive to produce.

This approach also suffers from a lack of flexibility when changing print mechanisms and communications interfaces. The board designer could improve extensibility by providing plug-in boards for communications and print-mechanism interfaces, but this solution still falls short when considering up-front development costs and time-to-market.

As soon as we consider designing a separate communications interface and print mechanism interface, the next logical alternative to designing the main board is purchasing one off-the-shelf. A number of PC/104 single-board computers on the market make good candidates for a substitution.

PC/104-based SBCs are compact versions of the IEEE P996 PC and PC/AT bus computers. These boards are generally useful for designs where mechanical room is not available for a traditional PC. While the cost of peripherals may run slightly higher, the hardware involved is electrically identical to standard PC peripherals in many cases, so you can use familiar DOS development tools in the project.

Alternatively, if space permits, you can use an off-the-shelf PC

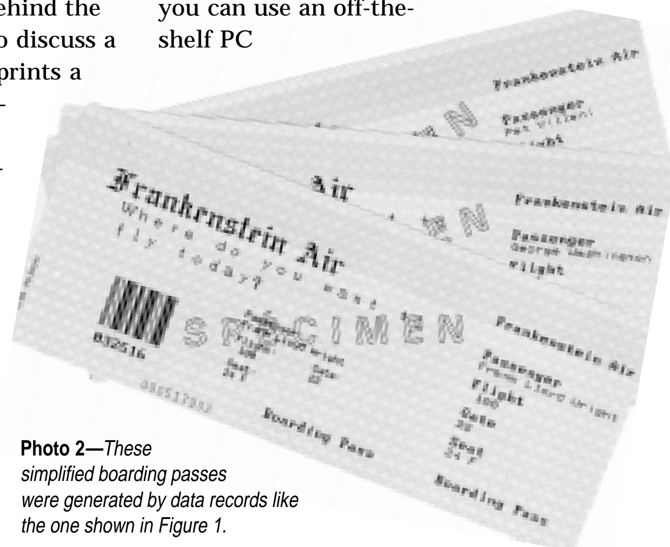


Photo 2—*These simplified boarding passes were generated by data records like the one shown in Figure 1.*

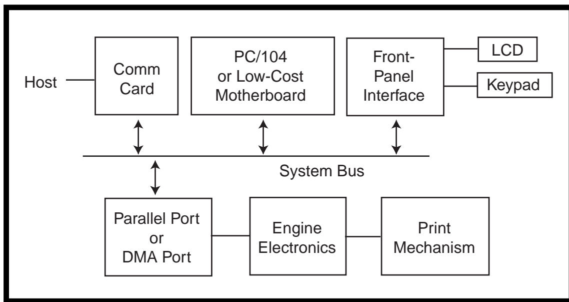


Figure 2—*The bus-oriented design is evident in the hardware architecture.*

motherboard, further reducing development and recurring costs. The choice is entirely dictated by packaging needs.

However, there are some tradeoffs made with this design. For example, you are limited in the choice of processors available with these boards. Typically, only 80*x*86-based processors are available in these form factors. This means your software developer may face the challenges of segmentation in real-mode 80*x*86 code. This may not seem significant at first, but consider the math.

Let's look at a document that is 4″ × 7.5″, with an image area of 3.5″ × 7″. If our resolution is 200 dpi, that's 700 × 1400 pixels for a total of 980,000 pixels.

Assuming a single bit per pixel, the image buffer area is 122,500 bytes—nearly 120 KB. Unfortunately, segmentation limits you to 64 KB for the simple offset-only read-modify-write cycles needed for imaging and forces you to more complex segment:offset imaging techniques.

In practice, the consequence could be as much as a 4× performance degradation. Using an 80386+ running in protected mode or 68k processor will improve on this but at an increased recurring cost.

Other design issues also must be addressed. For example, transferring data from the system memory to the engine may also place restrictions on the choice of processor boards.

You may want to design a custom engine that uses DMA for image transfer. This is part of most 80*x*86 boards, especially PC/104 boards, so it may not be a problem.

However, other low-cost boards may not include a DMA controller and may not have enough memory for an image buffer, making them unsuitable for this application anyway.

Another consideration is the type of print mechanism used. You can use almost any printing method—dot matrix impact, thermal transfer, direct thermal, inkjet, and so forth. Each method has some advantages and disadvantages.

The mechanism you choose is a function of the application. If the printer is in relatively harsh environments such as an airline terminal, which tends to be dusty and poorly ventilated, the impact of a dot matrix printer tends to generate a lot of paper dust.

You also need to think about consumables like ribbons or toner. Imagine being stopped while boarding a plane because they have to change a ribbon or add some toner—what a nightmare!

Therefore, I've always chosen direct thermal transfer for my designs. In direct thermal transfer, there are no ribbons or toner to contend with. The print head makes direct contact with the ticket stock. Heat is generated by individual elements on the head, causing salts coating the stock to change from translucent to black.

There are commercially available, low-cost mechanisms that can be used to implement direct thermal imaging. However, you must carefully consider these mechanisms because they may wear out sooner than some custom-designed mechanisms.

On the other hand, maybe that's a feature. Before you accuse me of "marketing talk," consider the repair issues.

A self-contained print mechanism is easier to replace for field service personnel than replacing a head or set of rollers. In the former, the entire mechanism is aligned at the shop or factory as opposed to aligning a head or setting roller pressure while crouching behind an airline counter.

One hidden issue for print mechanism choice is print speed. For example, direct thermal can be reasonable but may entail working with thermal history. Figure 3 graphs the paper motion under the print head.

Adjacent dots may affect the amount of energy required to change that dot from clear to black. Those adjacent dots in the same column each contribute the amount of energy.

In addition, the preceding dot in the same row also contributes because of the thermal time constants in the head. This is the effect known as "history" and may cause a dot to spread, bloom, or smear.

To overcome this, a typical technique is to break up the application of energy into a number of smaller, sequential "burns" and decrease the number of "burns" applied to any dot by dynamically changing the bit pattern for each column of dots.

This may require either special hardware or a faster processor to create the modified bit pattern.

Alternatively, you can print at a slower rate if your application can accept slower print speeds.

Another mechanism, such as laser ion-deposition, may be faster but at a higher cost. And, it requires consumables.

Again, the choice is dictated by your application.

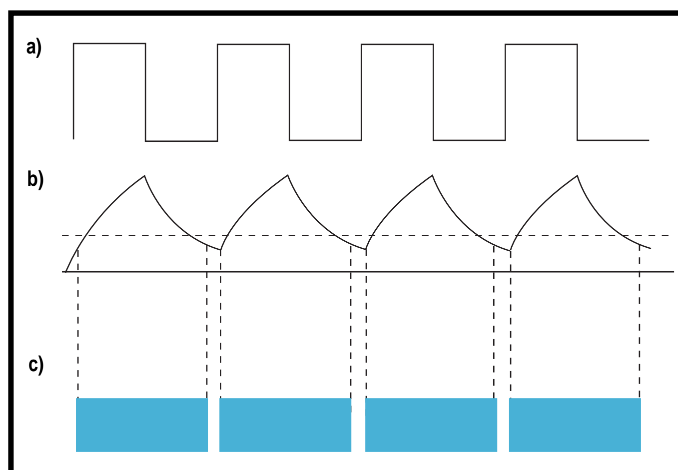


Figure 3a—*Pulses applied to the head cause an individual element to heat as shown in (b).* ***b****—Successive pulses do not allow the temperature to return to ambient temperature and cause more of the pulse of energy to exceed burn temperature. This causes successive dots (****c****) to appear to stretch and smear.*

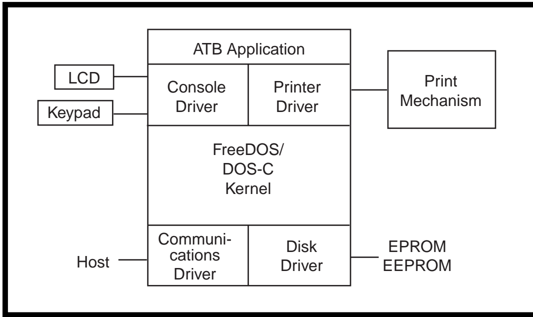


Figure 4—*The FreeDOS/DOS-C kernel sits between the various drivers in the software architecture for the boarding-pass printer.*

SOFTWARE ARCHITECTURE

Typical ATB applications are generally quite complex and need many communication modules and interface emulation modules, all needing different types of interface drivers. We've seen that our application may require different print mechanism drivers for the various types of stock that may be used by the printer or family of printers.

To meet these needs, let's look at an architecture that makes use of the embedded kernel for storage of multiple applications. Last month, I described the FreeDOS kernel and looked at the FreeDOS project as a source for royalty-free source code. Let's build on that.

Figure 4 represents a potential architecture for an ATB printer. As you can see, the ATB printer is designed to work on top of the OS. In this way, the combined application and OS appear as a seamless package to the outside world.

The kernel, combined with custom device drivers, becomes the primary interface to the hardware. We now use standard operations such as `write` to output the image to the print engine and `read` and `write` for interfacing to the front panel.

The heart of my ATB application is a small graphics package I call Light Weight Graphics (LWG). This package contains the simple set of APIs listed in Table 1.

It is also designed to be modular, breaking out the actual imaging to a graphics device driver. I use this to my advantage by doing development with a screen driver and then replacing it with a printer driver for the final application. In many ways, this is similar to the Windows Device Context abstraction.

Listing 1 shows a simple loop that monitors the data port and then images the ticket. It's simple yet powerful. You can easily replace the application portion and adapt the design to any airline specification.

The graphics package is initialized with a single call to `LWGinit()`. From there, you can select any font you

API	Description
BYTE *ParseIni(BYTE *pszBuffer, BYTE *pszSection, BYTE *pszEntry)	Parses `.INI` file
VOID LWGinit(VOID)	Initializes LWG package
VOID SetMargin(WORD wLeft, WORD wTop)	Sets page margins
WORD GetSystemFont(VOID)	Get default system font
WORD RemoveFont(WORD f_no)	Remove font from font cache
WORD LoadFont(BYTE f_name)	Load font into the font cache
WORD SelectFont(WORD f_no)	Select font to print/display with
VOID SetBkColor(WORD nColor)	Set background color
VOID SetFgColor(WORD nColor)	Set foreground color
VOID TextOut(WORD x, WORD y, BYTE *s, WORD mode)	Output string to printer/display
VOID BitBlit(WORD x, WORD y, WORD ch, struct fon fp)	Image single character
VOID Delay(WORD n)	Delay *n* seconds
VOID HorzLine(WORD x, WORD y, WORD len, WORD wid)	Draw horizontal line
VOID VertLine(WORD x, WORD y, WORD len, WORD wid)	Draw vertical line
VOID DrawLine(WORD x1, WORD y1, WORD x2, WORD y2, WORD (*f)(WORD, WORD))	Draw line in any direction using paintbrush
VOID Update(VOID)	Update display or print page
VOID ShutDown(VOID)	Terminate LWG package
VOID ClearDevice(VOID)	Clear display/printer by repainting background

Table 1—*The Light Weight Graphics (LWG) API is part of a simple, efficient graphics package that's suitable for embedded applications.*

wish with calls to LoadFont(). This call returns a font handle that is defined as a 16-bit integer.

This handle is used in subsequent calls to SelectFont() to set the font to be used by the text imaging functions or to RemoveFont() to delete the font from the font cache.

Text to be printed is handled by TextOut(), and lines can be drawn with DrawLine(). However, for barcodes, there are two optimized line-drawing functions—HorzLine() and VertLine().

When imaging is complete, a single call to Update() causes the image to transfer to the print engine. The process may be repeated by clearing the image buffer with a call to ClearDevice() and doing it all over again.

The graphics driver is separate from the imaging package. In LWG, the driver is typically a single file that exports a single data structure, GDriver, that contains data pertaining to document height, width, and resolution.

It also contains pointers to various driver entry points. This structure is an abstraction similar to a C++ class and is the method used to make the package work with various print engines.

The package also contains support functions to set color, if needed, and margins. It also contains a function to parse a Windows-like initialization. This function, ParseIni(), opens and reads a file, looking for a section contained in square brackets, and then returns the string starting with string=.

This additional customization enables the printer to be even more flexible by passing information to your application. Your application then uses this information to customize its operation such as selecting different font formats, communication protocols, and so on.

By this time, you're wondering why I'm putting a hard drive into a printer. I'm not adding any drive. I can replace the drive with a simple device driver that maps an EPROM or EEPROM to a pseudo-disk drive.

I've used this technique before and kept the FAT format. Although it adds overhead, all I had to do to create an EPROM was to copy the application to a floppy disk. Next, I read the floppy

Listing 1—*This sample application makes use of DOS-C and LWG. The entire application is coded in a few hundred lines, significantly reducing the resources needed.*

```
#include <stdio.h>
#include "portab.h"
#include "lwg.h"
#include "barcode.h"
#include "ui.h"

#define MAX_PASSNAME  32
#define  MAX_NUMBER  6
#define  MAX_SEAT  4
#define  MAX_FLIGHT  4
#define  MAX_GATE  4

VOID Image(VOID);
VOID CommInit(VOID);
static  WORD
  nLeftMargin = 0,
  nTopMargin = 0,
  nSysFont,
  nPassFont,
  nPassFont1,
  nOldEngFont,
  nThinFont;
static FILE *pInputDevice;
static struct Record{
  BYTE  szPassengerName[MAX_PASSNAME+1];
  BYTE  szRecordNumber[MAX_NUMBER+1];
  BYTE  szRecordSeat[MAX_SEAT+1];
  BYTE  szRecordFlight[MAX_FLIGHT+1];
  BYTE  szRecordGate[MAX_GATE+1];}
DataRecord;
VOID main()
{
  LWGinit();
  CommInit();
  nSysFont = GetSystemFont();
  nThinFont = LoadFont("s16x16.fon");
  nOldEngFont = LoadFont("oe7.fon");
  nPassFont = LoadFont("s6x8.fon");
  nPassFont1 = LoadFont("s8x8.fon");
  SetBkColor(WHITE);
  SetFgColor(BLACK);
  SetMargin(30, 0);
  FOREVER{
    if(OnLine()){
      COUNT nRead;
      nRead = fread(&DataRecord,1,sizeof(DataRecord),pInputDevice);
      if(nRead < 1){
        ShutDown();
        exit();}
      else{
        DataRecord.szPassengerName[MAX_PASSNAME] = '\0';
        DataRecord.szRecordNumber[MAX_NUMBER] = '\0';
        DataRecord.szRecordSeat[MAX_SEAT] = '\0';
        DataRecord.szRecordFlight[MAX_FLIGHT] = '\0';
        DataRecord.szRecordGate[MAX_GATE] = '\0';}
      ClearDevice();
      Image();
#ifdef DEBUG
      Delay(15);
#endif
      Update();}
    else{}}}

VOID Image(VOID)
```

(*continued*)

and converted it to Intel hex records. Finally, I downloaded it into an EPROM programmer. Simple and effective.

I've also written special device drivers that created a simple memory file system but now it involves adding code to the kernel, so you must keep GPL in mind. Either way, all your code goes into EPROM or EEPROM, eliminating the need to write firmware.

DEVELOPMENT ENVIRONMENT

It should be apparent that this design offers strong advantages for development. For example, you don't need special compilers to generate code for firmware.

You can use standard compilers that run under MS-DOS and Windows 95 to generate your code. You only need to make sure that your compiler can generate code for MS-DOS.

You also debug your entire application under MS-DOS, burn it into ROM, and do only final system testing on the host. No mess, no fuss.

In the sample code provided for this article, I created a project that accepts data from standard input and creates an image. This project includes two drivers: one for a VGA display and another for an HP laser printer.

The application reads standard input and looks for a start record. Once found, it reads each record and creates a document for the record. Check out both the VGA and laser-jet code. It will help you see how easy this type of development can be.

TAKING OFF

Now you've seen an application based on FreeDOS that takes advantage of the OS's standard system calls and device drivers. I only talked about intelligent printers, but you can extend the design to other areas.

For example, you can expand the front-panel keyboard to a full keyboard and make it into a modern teleprinter. If you remove the print mechanism and add ADCs, the same basic architecture can become a remote monitoring system or data-acquisition system.

Or, you can use the graphics package combined with a cable converter to make an intelligent set-top converter. The possibilities are limited only by your imagination.

Listing 1—*continued*

```
{
  SelectFont(nOldEngFont);
  TextOut(0, 56, "Frankenstein Air", X_INC);
  SelectFont(nThinFont);
  TextOut(20, 60, "Where do you want to", X_INC);
  TextOut(20, 80, "fly today?", X_INC);
  SelectFont(nSysFont);
  TextOut(220, 190, "Boarding Pass", X_INC);
  TextOut(400, 25, "Frankenstein Air", X_INC);
  TextOut(400, 190, "Boarding Pass", X_INC);
  TextOut(400, 60, "Passenger", X_INC);
  TextOut(400, 90, "Flight", X_INC);
  TextOut(400, 120, "Gate", X_INC);
  TextOut(400, 150, "Seat", X_INC);

  SelectFont(nSysFont);
  TextOut(40, 180, DataRecord.szRecordNumber, X_INC);
  vbar(DataRecord.szRecordNumber, 40, 100);
  SelectFont(nPassFont);
  TextOut(170, 100, "Passenger:", X_INC);
  SelectFont(nPassFont);
  TextOut(170, 110, DataRecord.szPassengerName, X_INC);
  SelectFont(nPassFont1);
  TextOut(400, 65, DataRecord.szPassengerName, X_INC);
  SelectFont(nPassFont);
  TextOut(170, 125, "Flight:", X_INC);
  SelectFont(nPassFont);
  TextOut(170, 135, DataRecord.szRecordFlight, X_INC);
  SelectFont(nPassFont1);
  TextOut(400, 95, DataRecord.szRecordFlight, X_INC);
  SelectFont(nPassFont);
  TextOut(250, 125, "Gate:", X_INC);
  SelectFont(nPassFont);
  TextOut(250, 135, DataRecord.szRecordGate, X_INC);
  SelectFont(nPassFont1);
  TextOut(400, 125, DataRecord.szRecordGate, X_INC);
  SelectFont(nPassFont);
  TextOut(170, 150, "Seat:", X_INC);
  SelectFont(nPassFont);
  TextOut(170, 160, DataRecord.szRecordSeat, X_INC);
  SelectFont(nPassFont1);
  TextOut(400, 155, DataRecord.szRecordSeat, X_INC);}

VOID CommInit(VOID)
{
  pInputDevice = stdin;
}
```

Pat Villani has 22 years of industry experience in both hardware and software, most recently developing firmware and real-time kernels. He currently works for a major computer company writing portions of OS kernel and system management tools. You may reach Pat at patv@iop.com.

SOFTWARE

Source code for this article is available via the Circuit Cellar Web site and <www.iop.com/~patv>.

SOURCES

PC/104 Specifications
PC/104 Consortium
(415) 903-8304
Fax: (415) 967-0995
www.controlled.com/pc104/consp1.html

IEEE P996 Specifications
IEEE Standards Office
(800) 678-4333
(732) 981-0060
Fax: (732) 981-0225
www.ieee.org

An Alarming Improvement

Jeff Bachiochi

Part 2: Assembly Language Takes the Race

Engineers are never satisfied! Jeff's been thinking about the alarm monitor he and Steve built and he wants to speed it up. Check in to see how a little assembly code and some utilities get things rolling.

have you ever finished a project only to wish you could have a second chance at it? Engineers are known for continually wanting to tweak things. Nothing's ever perfect.

Well, I'm taking the time to improve on the project Steve and I presented last month ("Gotcha! Alarming the Alarm System," *INK* 95). A couple routines in that BASIC-only program have been bothering me.

The local keypad and the LCD were manhandled by bit-manipulating digital I/O on the Domino's coprocessor. Similar to bit-banging a software UART, BASIC made communicating with the peripherals sluggish.

The BASIC language masked into the Domino has some terrific advanced features that make it extremely powerful. While the ease of programming in BASIC is great for the novice, embedded hooks enable the experienced programmer to call on assembly routines when the going gets rough (or slow).

The console serial port is used as the primary interface for user I/O. The `PRINT` statement sends data out the serial port, while the `INPUT` and `GET` statements gather data coming in the serial port.

Domino's BASIC command set has a pair of user commands that let you redirect the data path. On the input side, `UI0` tells the processor to accept data from the serial port, while `UI1` lets the processor get input data from a user-written routine encountered at a predefined location. On the output side, `UO0` sends the output data to the serial port, while `UO1` redirects the output to a user-written routine located at a predefined location.

Although the keypad and LCD are connected to the Domino's I²C coprocessor, they look like digital I/O to it, not a keypad or LCD. To talk with them using BASIC, each device must be manhandled in order to get a look at the keypad or write to the LCD.

The I²C communications are quick, but all the BASIC code that must be executed slows things down. So even though the alarm-monitor project didn't require fancy programming or speed, I cringed while waiting for the LCD to paint a message one character at a time.

Although that project is finished, I still feel there's something to be learned. Using these advanced hooks, I can make the directly connected keypad and LCD a pleasure to use, instead of a pain. And, I can speed up the whole shebang. But first, let me say a few words about two useful shortcuts.

OP-BYTES

BASIC-52 provides access to individual assembly-language routines located in its masked processor. Almost every function available as a BASIC command can be called directly, eliminating the interpreting, which is where the snails clog up the execution.

To use these assembly-language routines, place a value (which identifies the function) in the accumulator and call address 0030h. This method can really cut down on the code necessary to write powerful routines.

DOMINO UTILITIES

Because the original BASIC-52 was written to support EPROM devices and the Domino uses flash memory, new program-save algorithms were required. At the same time, other functions were added to support the novice programmer.

In fact, the I²C routines added to the utilities significantly expand the Domino's I/O capability. The utilities' code takes full advantage of the op-bytes

available through the BASIC interpreter. In the same way, I can use the op-bytes and utilities to write keypad and LCD functions.

USER INPUT

Here's how I was able to take advantage of both BASIC op-bytes and the Domino utilities. Note that I did have access to the utilities source code and could have skipped the op-bytes by placing the data being passed on the stack into the appropriate registers and jumping into the utilities code after the stack-removal routines.

However, if the utilities ever change, the jump into the utilities could change. By using the routine at its documented entry point, my routines stay compatible even if the utilities change in the future. Check out the pseudocode in Listing 1 as I go through it.

The BASIC command `INPUT` collects variable or string data until a <CR> is received. It then continues program execution.

The BASIC command `GET` looks for a single character and then continues with program execution. Unlike `INPUT`, `GET` doesn't suspend the program's execution if no input is available.

These commands use two routines. First, a status-check routine checks the serial port's input buffer to see if a character has been received. The carry flag is set if a character is ready. Otherwise, the carry is cleared. A second routine gets the character and places it into the accumulator.

When the input is redirected using `UI1` to the user's input routines, a jump is made to 4030h or 4033h in code space. At these locations, you need to place a jump to your own code.

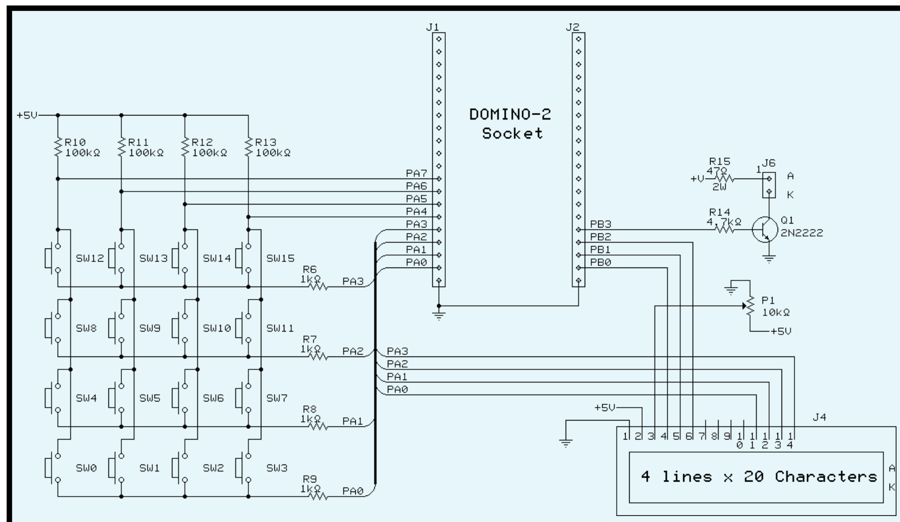


Figure 1—*Simple interfacing keeps the costs low but places higher demands on the software.*

The standard console input has a flag you can test to see if a character has finished coming into the UART. The keypad has no built-in flag we can check with. We must go through the same routine originally written in BASIC to determine whether a key was pressed and to find out which one it was.

Figure 1 shows that the four column inputs have weak pullups to provide a logic 1 on the upper nibble inputs in the idle state. The rows are driven from the lower four bits. Normally, a high on these outputs doesn't change any of the upper nibble inputs even if a key is pressed.

To scan the keypad, each row is driven low, one at a time. While the row is low, the column's inputs are read. Any key pressed passes the row's logic 0 to the appropriate input. Series resistors (or diodes) in the row outputs prevent multiple keys from directly shorting out two row drivers.

Note that the only way to see if a key is pressed is to go through the same steps that determine which key was pressed. Therefore, the two input routines can be pretty much the same.

The `get_key` routine must return the pressed key's value in the accumulator. I'll start by describing that one.

When `UI1` is executed, the processor redirects the program flow to the vector location 4033h, which is right in the middle of RAM. MTOP should be set below this address (I usually set it to 3FFFh).

Since the program now expects to execute an assembly-language operation at this address, you need to place a short or long `JMP` here that again redirects program flow to the beginning of your routine. I `ORG`'d the code to start at 4030h, the first of the jump vectors. The routines follow these jumps.

To use the I^{2}C routines that set and read the digital I/O of the coprocessor from BASIC, you need to push an I^{2}C address, a register number, and a value on the stack. Then, a call is made to address F128h, where the utilities perform the I^{2}C bit-banging routines to communicate with the coprocessor. To use the same call to the utilities from assembly language instead of BASIC, you must again place the appropriate values on the stack.

One small hitch. The BASIC `push` places a floating-point equivalent of

Listing 1—*To keep everything straight, 16-bit values are converted to floating point by op-byte routines. These then become a setup for the calls to the utility routines.*

```
Keypad
  Initialize column counter, column mask, and key deposit
  Loop
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I2C retrieve registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I2C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
  Check byte
    If no low bits (key pressed)
      Decrement column count
      If more columns goto Loop
  Exit
```

the parameter on the stack (six bytes for each parameter). Even though all parameters are 16-bit words, be sure to place the floating-point equivalent on the stack just the way BASIC did or the utilities won't speak the same language when they're removed from the stack.

This task is done quite easily without having to write the conversion, thanks to the op-bytes. The initialization of the coprocessor's digital I/O becomes just a bunch of register setups and op-byte calls followed by utility calls to handle the I²C bit banging. Three temporary registers handle the routine's variables—a column counter, column mask, and key-down input value.

During the column loop, the column mask is sent to the coprocessor and the row values are read back from it. Since the utilities use the same temporary registers I need, they must be pushed and popped on either side of the calls to the coprocessor (missing that little tidbit caused me some pain during development).

Knowing which column is active and the value returned enables you to determine if a key was pressed and which one it was. The column loop repeats for all four columns if no key press is detected.

Since this routine deals with both the detection and identification of keys pressed, I divide the remaining routine into two functions. The first branch sets the carry flag if a key is detected and clears it if no key press was found. The second branch takes the key value (1–16) and adds 2Fh to make it a printable character.

Then, this code makes a compare to what was stored in the last sample permanent register. If it's the same, it zeros out the accumulator. Otherwise, it places the key value into the accumulator. (This step prevents multiple detections of the same key press.) After some register cleanup, you're done.

When BASIC wants to know if a character is ready, program execution is redirected to the status-check routine's vector at 4036h. Here, you must place a short or long jump redirect execution to the beginning of your routine.

`key_stat` calls the `key_get` routine with two small additions. First, a flag

Listing 2—*LCD print routines make heavy use of both op-bytes and utility routines.*

```
LCD
  If character is '0'
    Clear RS output bit
      call BASIC op-byte (16-bit integer to floating point)
      call utility (I²C retrieve registered byte)
      call BASIC op-byte (floating point to 16-bit integer)
      call BASIC op-byte (16-bit integer to floating point)
      call utility (I²C send registered byte)
      call BASIC op-byte (floating point to 16-bit integer)
    Goto Exit

  If character is 'FF'
    Set RS output bit
      call BASIC op-byte (16-bit integer to floating point)
      call utility (I²C retrieve registered byte)
      call BASIC op-byte (floating point to 16-bit integer)
      call BASIC op-byte (16-bit integer to floating point)
      call utility (I²C send registered byte)
      call BASIC op-byte (floating point to 16-bit integer)
    Goto Exit

  Clear R/W bit
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C retrieve registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
  Send upper nibble of character
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
  Set E bit
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C retrieve registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
  Clear E bit
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C retrieve registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)

  Send lower nibble of character
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
  Set E bit
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C retrieve registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
  Clear E bit
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C retrieve registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
    call BASIC op-byte (16-bit integer to floating point)
    call utility (I²C send registered byte)
    call BASIC op-byte (floating point to 16-bit integer)
  Exit
```

is set so key_get knows it is being called from key_stat. Second, the accumulator must not be altered, so it is pushed prior to the call and popped after returning from it.

Later on, I'll compare the code space and execution times of the BASIC routines with the assembly routines, but first, let me give you a look at the output side of things.

USER OUTPUT

Again, I make extensive use of both the BASIC op-bytes and Domino utilities routines. Most routines here do a read-modify-write to change individual bits (see Listing 2), so these routines can't write incorrect data to any I/O not being directly used by the keypad or LCD.

Although it may not look like it, the user output routine is much less complicated than having to scan and sample the columns and rows of keys making up the keypad of switch contacts. After all, with the output, you only have to strobe out the data to a parallel connected LCD.

Well, that's marketing talk. The engineer's mind is questioning nibble transfers, busy status, and control registers. While none of these items is insurmountable, the last is cause for concern.

The standard console output routine merely places data to be transmitted in the UART's buffer and—whoosh—out it goes. U01 directs program execution to address 4030h, where a jump vector will again redirect it to the user's output routine.

Once the LCD is initialized by some means of writing to the LCD's control register, the redirected execution can write to its data register. Now, all characters just pop onto the LCD, one after another.

But, to position the cursor somewhere else or to clear the screen, you need to be able to talk to the control register. A UART doesn't need this complexity, but the LCD does. How can the output routine process both data and control information without being confused about what it's handling?

My solution is similar to using the special character modes in a word-processor document. To force a word processor into (or out of) this mode (be it underline, bold, superscript, etc.), you use a special character—one that isn't ordinarily used as a text character.

In this program, the characters 0 and 255 are not necessary alphanumeric characters, nor are they useful control-register data. So, I used them as mode signals.

If a 0 is received by the user output routine, the RS (register select) bit is cleared. So, characters are interpreted as control-register data. If a 255 is received by the user output routine, the RS bit is set and characters are interpreted as display data.

Neither of these characters is passed to the LCD. They can be embedded into print strings to add screen control within the print data.

BASIC hands off characters to the user routine via PRINT. Each character goes through the user routine, which interrogates it. Once the character is broken into nibbles and strobed into the LCD, the routine exits. An eight-bit LCD mode would be faster because

	Keypad	LCD
BASIC	300 ms	140 ms
Assembly	28 ms	44 ms

Table 1—*Assembly routines offer significant speed increases over interpreted BASIC.*

the data bus is twice as wide, but I wanted to leave four I/O bits free for other things.

To slim the routines, an initialize-ports routine sets up the direction registers for Port A and Port B and places the outputs into an idle logic state. If the routine is passed a `0` or `255`, then the RS output bit is set or cleared accordingly and the routine exits without passing any data to the LCD.

The RS output bit instructs the LCD to direct data to either the control registers or display. Since screen-position data can be easily placed with display data into BASIC's `PRINT` statement, the user can position text anywhere on the LCD screen.

FITTING IT TOGETHER

I now have this chunk of assembly code that needs to get pushed into RAM starting at 4030h. The Intel hex file can't be easily loaded.

I like using `DATA` statements to `XBY` the data (code) into the appropriate locations. To make the data entry easy, I wrote a GWBASIC program to read in the hex file and output BASIC-formatted `DATA` statements that can be downloaded along with the rest of a BASIC program into the Domino.

As a plus, other BASIC statements are automatically added to provide a for-next loop that places data into the Domino's RAM at a location based on the hex files' instructions. When the BASIC program is run on the Domino, a call is made to this routine, which puts the assembly code into place before the rest of the program needs it.

TIME TRIALS

Table 1 shows the execution times for the keypad's scan rate and the LCD's character rate. The communications with the I²C coprocessor require only about 4 ms total for a keypad scan or character transmission.

Since both BASIC and the `UI1` and `UO1` routines use these communications, the time savings is the direct difference between the BASIC and assembly-language times. The LCD increase was only 3:1, but the keypad routine increased the scan speed tenfold. Although these speed increases aren't overly dramatic, the easy of use alone is worth the time investment.

In the total-BASIC version, the cursor-position and character sequences had to be sent and strobed into the LCD a nibble at a time. Then, the character string had to be parsed, split, and strobed into the LCD by the nibble as well.

When you use `UO1`, this task is handled in the background and you only need `PRINT CHR(0), CHR(64), CHR(255),"Hello World"` to position the cursor to column one of the second row and display a message.

To improve the speed even more, you could use a smart I²C peripheral, such as a serial LCD. This type of device offloads all of the display control to a local processor. However, in low-budget operations, the minimum-hardware approach often saves real dollars.

In this case, I was able to improve speed and ease of use, while maintaining a certain level of harmony between the embedded BASIC interpreter and the extended assembly-language utilities—and I didn't add to the parts cost.

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

SOURCES

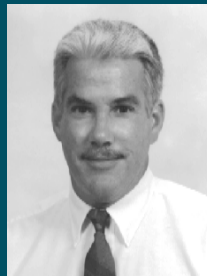
Avocet 8051 Assembler
Avocet Systems, Inc.
(207) 236-9055
Fax: (207) 236-6713
www.avocetsystems.com

Domino2
Micromint, Inc.
(800) 635-3355
(860) 871-6170
Fax: (860) 872-2204
www.micromint.com

SILICON UPDATE

Tom Cantrell

The Micro Price is Right



Motorola's launch of an 8-bit OTP for less than 50¢ takes Tom down memory lane. Although the 68HC705KJ1 uses an 'HC05 core, it adds a surprising amount of I/O and support logic. Hear the call of another 8-bit war?

o sooner did I finish last month's '51 flashback than a press release from Motorola drops on my desk announcing "what is believed to be the world's first 8-bit OTP microcontroller available for less than 50 cents." Well, the price may be new, but the 'HC05—in this case, the 68HC705KJ1—is as old as the hills.

Worth covering? I might be accused of trying to relive my youth except for the fact that the old-timers still make up the bulk of 8-bit shipments. Also, I try to remind myself that each new crop of engineers is less likely to remember these chips that trace their heritage 20–25 years back to the dawn of micros.

In fact, I realized with a twinge of mid-life crisis that stuff like disco, bell bottoms, and *The Brady Bunch* is apparently riding some kind of nostalgia wave (I guess you had to be there to know better), so why not '51s and '68s?

HISTORY LESSON

As best I can recall, it was the early '70s when the micro wars began in earnest. Intel, having pioneered with the 4004 and 8008, was moving quickly out of the blocks with the 8080 when Motorola responded with the 6800.

It's interesting to reflect on the showdown's specifics in light of subsequent events. Fact is, the 6800 was technically superior to the 8080.

And we're not talking technical minutiae, but rather major functional advantages. For instance, the 6800 was +5-V-only operation while the 8080 required three supplies (+5, +12, –5 V). Worse, most 8080 setups called for at least two other chips—the 8224 clock generator and 8228 bus controller.

As for architecture and programming, the 8080 was rather eccentric. All those weird nonorthogonal instructions, registers, and addressing modes. Can anyone remember what the heck instructions like `XTHL` and `SPHL` do? In fact, the story goes that Intel engineers Faggin (later to start Zilog) and Mazor kept adding instructions until the normally circumspect Shima cried, "Enough!"

SIMPLER TIMES

By contrast, the 6800 programming model was much simpler. As you see in Figure 1, it lives on little changed in today's '05. It's real easy to describe and understand.

Start with the fact there are five basic types of instructions: register/memory, read-modify-write, jump and branch, bit manipulation, and control.

Most register/memory instructions (e.g., `ADD`, `CMP`, `OR`, `LD/ST`, and even `MUL`) use two operands—one a memory location and the other either the accumulator (`A`) or index register (`IX`).

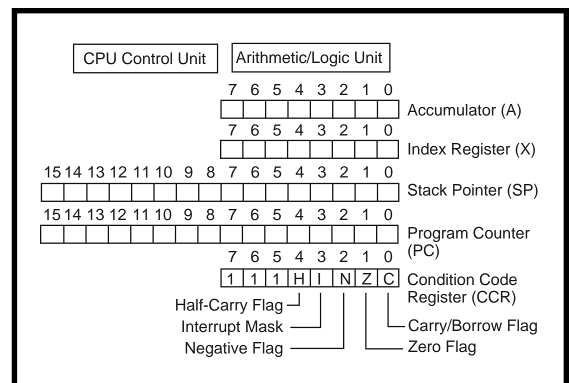


Figure 1—*Other than deletion of a second accumulator and limited memory addressing, the 'HC05 programming model is little changed from the '70s-era 6800.*

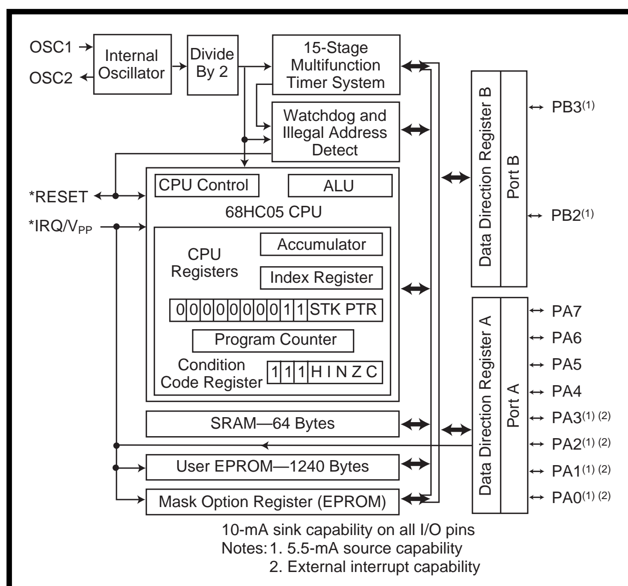


Figure 2—*The 68HC705KJ1 combines the 'HC05 core with 1.2-KB EPROM, 64 bytes of SRAM, timer, and parallel I/O—all in a 16-pin package.*

Read-modify-write instructions, including shifts, rotates, increment, decrement, negate, complement, clear, and so forth, work not only on registers but also directly on memory.

I know CISC is out of favor, but for simple ops, it's quicker and easier to modify a memory (or memory-mapped I/O) location directly than to load it into a register, twiddle it, and then store it back. As a precaution, avoid using read-modify-write instructions on memory-mapped I/O locations with write-only bits.

Conditional branches (plus 127, minus 128 bytes) query the condition codes (carry, half-carry, zero, etc.) as expected. There are also variants that branch based on the state of a bit in memory or the *IRQ pin. For going beyond the short displacement, an unconditional `JMP` is provided that can reach anywhere in the program space.

Similarly, subroutine calls are handled with unconditional branch (`BSR`) and jump (`JSR`) instructions. They're just like regular branches and jumps, except they push the return PC on the stack.

The `SWI` (software interrupt) executes the equivalent of a subroutine call via a dedicated vector location, except stacking all register contents instead of just the PC. Two variants of return (`RTS` and `RTI`) unstack accordingly.

Bit operations include the ability to set, clear, or branch on the state of any bit in the bottom 256 bytes of the address space. Specific instructions set and clear the carry (`C`) and interrupt mask (`I`) bits in the condition code register.

Perhaps the most noticeable compromise (shared with many popular 8-bit chips) is restricted access to the stack. Only one instruction directly modifies SP (`RSP`), and it merely resets the stack pointer to the power-up default. It's best to use SP as the hardware stack and synthesize a software stack with `X`, assisted by instructions (`TAX` and `TXA`) that move data between `X` and the accumulator.

It may not be RISC, but the addressing modes are simple enough. The datasheet says there are eight modes, but it's really only five if you combine like variations. For instance, there are three versions of indexed addressing mode with 0-, 8- and 16-bit offset.

Similarly, there are 8- and 16-bit versions of direct addressing. Otherwise, there's inherent (i.e., argument address specified in the op-code, as with `TAX` and `TXA`), immediate (8-bit), and relative (which is only used for branches).

MINI-MICRO

Clearly, the 'HC05 is one of the simplest and easiest to understand chips ever. Motorola may cut the 'HC05 price to the bone, but it won't be by trimming any architectural fat.

Feature	Option
COP watchdog timer	Enabled or disabled
External interrupt triggering	Edge-sensitive only or edge- and level-sensitive
Port A *IRQ pin interrupts	Enabled or disabled
Port pull-down resistors	Enabled or disabled
Stop instruction mode	Stop mode or halt mode
Crystal oscillator internal register	Enabled or disabled
EPROM security	Enabled or disabled
Short oscillator delay counter	Enabled or disabled

Table 1—*The Mask Option Register is a special byte of EPROM that defines key operating characteristics of the watchdog timer, interrupts, ports, and oscillator.*

For most embedded micros these days, the CPU core only consumes a fraction of the die area. To paraphrase that old political wisdom, a few million transistors here (for memory) and a few million there (for I/O) and pretty soon you're talking real silicon.

So it goes with the 'KJ1, which packs the 'HC05 core in the silicon equivalent of a plain brown wrapper. It's best summed up by reviewing the block diagram in Figure 2.

Memory-wise, the 'KJ1 comes with 1240 bytes OTP EPROM and 64 bytes of RAM. Yes, it's minimalist, but not ludicrously so. I figure you can cram 10 pages or so of ASM or even a few pages of C (as long as you stay away from floating point).

A clock input running at up to 8 MHz (5 V, 4.2 MHz at 3.3 V) is divided by two on-chip, which yields a 4-MHz cycle time. Since instructions take from two to six cycles (except `MUL`, which takes 11 cycles), that works out to just about a 1-µs instruction average. If you want something faster, be prepared to pay more because there's no such thing as a free lunch—or free MIPS either, for that matter.

Though the clock setup eschews fancy PLL schemes found on higher priced chips, it is workmanlike. Options include the usual crystal, external CMOS input, and ceramic resonator.

There's also a special RC version of the chip that reduces external components to just a resistor, which itself can be eliminated if you're willing to accept whatever clock the chip delivers, typically between 1 and 3 MHz depending on voltage, temperature, and manufacturing variations. There's also an LC version that runs at 32 kHz off a watch crystal.

Low-power operation is largely a matter of clock control, via `STOP` and `WAIT` instructions. `STOP` completely shuts off the oscillator and thus the CPU and peripheral (e.g., timer, watchdog) clocks. Power consumption is cut to a few microamps, and CPU RAM and register data are retained all the way down to 2 V.

However, the only way to wake up is by external interrupt or reset input. An optional oscillator stabilization delay (see Table 1) holds the chip in reset for a few thousand clocks after wakeup.

Wait mode keeps everything except the CPU clock running. So, in addition to external wakeup, a watchdog reset or timer interrupt can provide an internal wake-up call. This cuts already low-active run-mode power consumption (<10 mA) by more than a factor of two.

I/O U

Considering the price and pin-count constraints (see Figure 3), the 'KJ1 comes with a surprising amount of I/O and support logic. For instance, there's both a RESET pin and an external interrupt input (IRQ, which doubles as a V_{PP} EPROM programming voltage). Either or both of these are sometimes sacrificed on other penny-pinching micros.

Reset includes a power-on circuit and Schmitt trigger input. It also is driven in response to an internally generated reset via the watchdog or—a plus for the safety conscious—if the MCU attempts to execute code at an illegal address. However, it takes a bit of clever software to figure out the origin of a particular reset because there are no cause bits and everything funnels through one reset vector.

A single 15-stage ripple counter is preceded by a prescaler that divides the internal clock by four (e.g., 1-µs timebase with 4-MHz internal clock). The first eight stages function as an 8-bit timer with maskable overflow interrupt.

The next four stages, preceded by some extra dividers, generate what's called a "real-time" interrupt with four options—16k, 32k, 64k, and 128k clock cycles. The chosen option defines the watchdog time-out period (one-eighth the real-time interrupt rate).

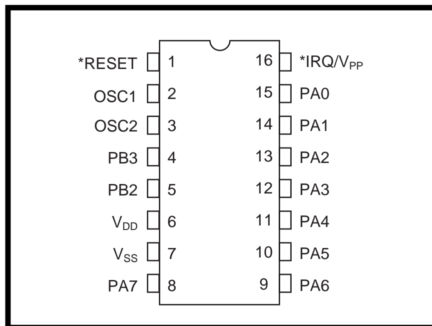


Figure 3—*Although it has only 16 pins, the 'KJ1 includes handy signals like *RESET and an external interrupt (*IRQ) along with 10 bits of I/O.*

Both timer interrupts (overflow and real-time) go through a single interrupt vector. To determine the source, simply check the corresponding interrupt flags in the timer status register.

That leaves 10 pins organized as one 8-bit I/O port (Port A) and one 2-bit I/O port (Port B). Each bit is programmable as input or output.

When configured as inputs, each line has an optional pull-down resistor. Note how one of the mask option register bits, SWPDI (software pull-down inhibit), disallows software enabling of the pull-down resistor. As outputs, they can handle LEDs or transistors with up to 10 mA per pin.

Mask option register setting, PIRQ, enables four pins of Port A (PA0–3) to function as additional external interrupt inputs (see Figure 4). Another mask option register bit, LEVEL, configures all external interrupts (i.e., IRQ and, if enabled, the PA lines) as either edge-only or edge- and level-sense. However, the polarity of each source is fixed (active low for *IRQ, active high for PA0–3).

LESS IS MOORE?

Moore's law, and the entire IC revolution, is often summed up as "more for less." However, it's really more subtle as in "more or less for less or more."

On the desktop, Moore's law historically means getting more chip for the same bucks as last year. This makes sense since it takes as many MIPS and megahertz as can be mustered to boot the latest bloatware.

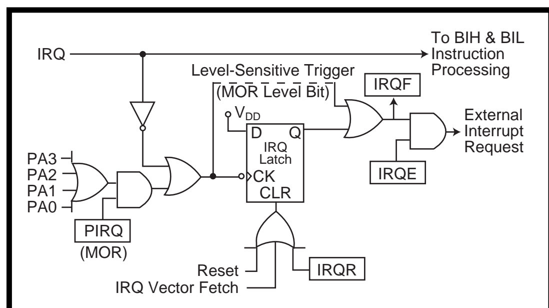


Figure 4—*In addition to the dedicated *IRQ pin, four pins of Port A can be configured as interrupt inputs. The* `BIH` *and* `BIL` *instructions enable fast and easy monitoring of the *IRQ pin in software.*

The embedded world is a bit different. Much the same scenario, more performance at a given price, is seen in the emergence of 32-bit chips, fancy RTOSs, and IP stacks, and so on. However, there's a huge segment of the market that wants the cheapest possible micro, even if it can barely compute its way out of a paper bag.

The name of the game is "how low can you go?" and I think 50 cents is pretty darn low. Remember, besides the die, they package, assemble, test, and ship, not to mention pay salaries.

A 50-cent micro sounds grand, but heed this caution: the low-cost OTP micro market is periodically wracked with shortages in which the latest darling suddenly disappears, inciting panic in the food chain.

If you ever took an economics class, maybe you remember there really is no such thing as a shortage—just a lack of means or the will to pay enough to balance supply and demand. When a capacity crunch hits, will IC suppliers start wafers filled with your 50-cent chips or someone else's $5 ones?

Chip companies long ago learned the folly of the "lose money on each one but make it up with volume" strategy. So tell your purchasing department to go easy. The "I got such a great price, they won't deliver" strategy is a loser.

My advice: Don't be greedy, but do take advantage of econo-chips like the 'KJ1. They may not do much more than chips of yore, but they do it for a lot less. ■

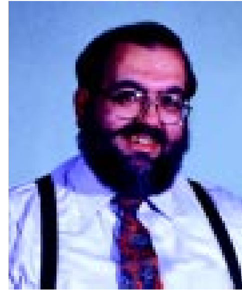
Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for more than ten years. You may reach him by E-mail at tom.cantrell@circuitcellar.com, by telephone at (510) 657-0264, or by fax at (510) 657-5441.

SOURCE

68HC705KJ1
Motorola
(512) 328-2268
Fax: (512) 891-4465
sps.motorola.com/csic/

PRIORITY INTERRUPT

It's All in How It's Done



Recently, a small manufacturing company approached me about designing an embedded controller to replace one they currently used. The present unit and its operating software were purchased as an off-the-shelf commercial product. When the OEM product doubled in price, the company simply decided to eliminate the supplier and make an equivalent unit. They asked me if I would reverse engineer the whole thing for them.

I made up a plausible excuse about having too many design jobs in the works and gracefully declined the opportunity. The task had some obvious challenges, but experience has taught me that reverse engineering involves considerably more than just technical issues.

On the surface, I don't have a philosophical objection with reverse engineering. Properly done, it's completely legal. In fact, evaluating a competing commercial design is an established practice in any new product design or product improvement. Certainly, everyone has heard the story about the BMW plant with all the disassembled Lexus cars in the corner. Unfortunately, in today's litigious society, designers and consultants have to beware of becoming so mesmerized by the technical challenge that they get oblivious to important legal issues. In the electronics industry, producing a clone of a piece of hardware or software is an established practice. It's using the proper technique to make a work-alike clone design that's the real issue.

First of all, be aware that a commercial product can be simultaneously covered by patent, trade secret, copyright, and contract laws (God bless those lawyers!). For reverse engineering to be legal, it must be legal under all these different laws. Here's what I typically watch out for.

Frankly speaking, reverse engineering won't help you as a defense against patent infringement. If your clone still infringes a patent, it won't make any difference what technique you employed to make it. The only solution here is to make sure that your design doesn't still include all the elements of the patented device.

As far as trade secrets go, it's more often an issue of misappropriation than reverse engineering. If you've got super taste buds and legitimately arrive at the formula for Coca-Cola, it should be defensible. However, if a disgruntled employee E-mails it to you, you had better think about the consequences before getting into the soft-drink business.

Copyrights are the primary protection for the software industry. It is permissible to dissemble object code in order to understand the functional operation of programs, but any effort should have a substantial paper trail showing how any clone was designed. Most importantly, the people responsible for disassembling code can only use it to produce a written specification. It should be a completely separate design team that uses that spec to create the new work-alike product. This was how the PC BIOS got into all those clones.

Finally, contract law seems to be the way most software companies fill the legal loopholes. These contracts usually contain so much legalese they could fill a law book. Perhaps you've noticed the shrink-wrapped impolite and severely worded licensing agreements on most software packages: "By opening this wrapper, I agree to…(give my first born, etc., etc.)…" Normally, whatever a contract says goes. The only exceptions are edicts that would be challenged under a standard contract defense.

The specific intention of a software license is to prohibit the licensee from disassembling or decompiling distributed code and to keep the licensee from discovering any unique programming techniques or secrets. In my opinion, overstressing the threats is counterproductive. There is also some argument whether banning decompilation can really be upheld.

Interestingly, microcomputer chip layouts can't be patented or copyrighted. They aren't considered novel enough for patent protection, and their utilitarian nature and application makes them ineligible for copyright protection. That doesn't mean they're an easy knockoff, however. To prevent chip piracy, manufacturers frequently use manufacturing techniques that make tracing the microcircuitry very difficult, as well as including nonfunctional patterns that help identify a copied layout. Reverse engineering is legal—copying isn't.

Financial reward is the obvious incentive that drives protection as well as innovation. Finding a middle ground that accommodates free expression without denying profit motive requires a careful balance between property rights and fair use.

You certainly don't have to get uptight every time someone mentions decompiling code or cloning a piece of hardware. In my experience, the copyright police aren't going to kick in your door if you disassemble Windows 98 "to better understand it." However, you can probably expect Bill Gates and about a hundred lawyers to personally get in your face if you try to sell what you find.

steve.ciarcia@circuitcellar.com