

EMBEDDED PC
MONTHLY SECTION

CIRCUIT CELLAR

INK[®]

THE COMPUTER APPLICATIONS JOURNAL

#99 OCTOBER 1998

DIGITAL SIGNAL PROCESSING

Getting Around the Nyquist Limit

Doing Filters in Software

Signal Conversion Basics

PIC-Based
Graphing Data Logger



\$3.95 U.S.
\$4.95 Canada

TASK MANAGER

Not Exactly a Binary World



Everybody's got their own way of looking at things. Fortunately—yes, fortunately—we don't all have exactly the same view of the world. It's also pretty clear that *INK* caters to a select audience. Not everybody, nor even every engineer, you meet designs or works with embedded computer systems for a living. But here's my point: Even within our narrow slice of the engineering universe, there are plenty of different perspectives. It's not all black and white, ones and zeros.

Over ten years ago, when Steve started *INK*, he was crafting a publication for hands-on engineers. And while a significant percentage of readers have been there since Day 1 (and among them will be those who call themselves experimenters as well as others who will vilify me for even suggesting it), *INK* has offered such an excellent range of editorial material over the years that its appeal now runs from systems designers in multimillion-dollar companies, to EE university students, to the engineering entrepreneur starting a brand-new company (that one day may in fact become one of those multimillion-dollar powerhouses), and beyond.

INK finds itself in a unique position, and faced with a unique problem: how to meet the needs of such a varied group. And I do mean *needs*. Nobody here is interested in putting together a magazine that gets tossed in the Freebies-To-Be-Read-Someday (yeah, right) pile. You pay for *INK*; you should get a magazine you want to read.

But, of course, there's only so much that can fit in a print magazine each month. That's why Design Forum is on *INK*'s web site. It's still in its infancy, but you know, sometimes, the more you give, the more you get. *INK* gives you Internet access to design hints, feature articles, abstracts submitted to *INK*'s design contests (most recently, Design98, cosponsored by Microchip), as well as monthly columns—and when you get inspired, you give back. Becoming an active part of the larger community that *INK* is fostering guarantees you'll find what you're looking for.

Design Forum is just one aspect of the growth here at *INK*. A new editorial advisory board has started, with three experienced engineering professionals—Ingo Cyliax, Norman Jackson, and David Prutchi—keeping their eyes and ears open for the kinds of editorial you're interested in. And although Ken moved on a couple months ago, he didn't go all that far: he still proofreads every article and helps with Design Forum editorial. Janice, too, has shifted roles. After doing a stellar management job for the past two years, she is now our Project Editor, helping with Internet editorial, as well as serving as a resource for special projects.

So, while *INK* is committed to its focus on providing high-quality engineering editorial, it's clear that we are determined to offer you a wider range of opportunities to experience it. No question: in the computing world, ones and zeros are a big part of the picture, but there's a lot of value lurking in the middle. Take advantage of it.

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR[®]

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Sue Skolnick

MANAGING EDITOR

Elizabeth Laurencot

CIRCULATION MANAGER

Rose Mansella

TECHNICAL EDITOR

Michael Palumbo

BUSINESS MANAGER

Jeannette Walters

WEST COAST EDITOR

Tom Cantrell

ART DIRECTOR

KC Zienka

CONTRIBUTING EDITORS

Ken Davidson

Fred Eady

ENGINEERING STAFF

Jeff Bachiochi

NEW PRODUCTS EDITOR

Harv Weiner

PRODUCTION STAFF

Phil Champagne

John Gorsky

PROJECT EDITOR

Janice Hughes

James Soussounis

EDITORIAL ADVISORY BOARD

Ingo Cyliax

Norman Jackson

David Prutchi

Cover photograph Ron Meadows—Meadows Marketing

PRINTED IN THE UNITED STATES

ADVERTISING

ADVERTISING SALES REPRESENTATIVE

Bobbi Yush
(860) 872-3064

Fax: (860) 871-0411
E-mail: bobbi.yush@circuitcellar.com

ADVERTISING COORDINATOR

Valerie Luster
(860) 875-2199

Fax: (860) 871-0411
E-mail: val.luster@circuitcellar.com

CONTACTING CIRCUIT CELLAR INK

SUBSCRIPTIONS:

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

GENERAL INFORMATION:

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

AUTHOR CONTACT:

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

For information on authorized reprints of articles,
contact Jeannette 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 responsibility 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** **X-Y Graphing Data Logger**
Alberto Ricci Bitti
- 24** **Time-Domain Filter Simulations Using C++**
Glenn Parker
- 30** **Breaking Nyquist—Post-Sampling Antialiasing**
Gerard Fonte
- 36** **Digital Frequency Synthesis**
Tom Napier
- 68**  **MicroSeries**
Digital Processing in an Analog World
Part 1: Basic Issues
David Tweed
- 76**  **From the Bench**
MIDI
Part 1: It Ain't Just for Music Anymore
Jeff Bachiochi
- 80**  **Silicon Update**
MegaMicro Card
Tom Cantrell

- Task Manager** **2**
Elizabeth Laurençot
Not Exactly a Binary World
- Reader I/O** **6**
- New Product News** **8**
edited by Harv Weiner
- Advertiser's Index/
November Preview** **95**
- Priority Interrupt** **96**
Steve Ciarcia
Banking on Bugs

INSIDE ISSUE 99

EMBEDDED PC

- 44** **Nouveau PC**
edited by Harv Weiner
- 48** **Networking with DeviceNet**
Part 2: A Weather-Station Application
Jim Brady
- 55** RPC **Real-Time PC**
The Need for Speed
RTOS and PC/104
Ingo Cyliax
- 61** APC **Applied PCs**
RF and Micros
Part 2: A Low-Power System
Fred Eady

READER I/O

AN ANALOG EYE ON DIGITAL DESIGN

I am an analog-oriented EE, and Mike Smith's article ("Unplanned Calibration Errors in Embedded Systems," *INK* 96) gave me an appreciation of the design considerations for a software program for a digital instrument. One would design an analog instrument to handle fail-safe conditions for all potential situations. The software code, its syntax, and the compilers used to mechanize code handling add layers of potential ambiguity.

In analog design, the simplest approach is usually the best, and I expect that it's likewise in digital design. I think your examples were picked to be a demonstration for your article, and it worked.

As an analog guy, I would be more likely to enter the correction factor/calibration error for the transducer at a summing junction, with the reference trip level for the fail safe at the low/high trip, to generate a signal to be subsequently summed with that of the conditioned transducer signal to generate a trip alarm.

I would think that, in a software design review, the software would be simpler, more reliable, and faster if

the static correction factor were applied as the digital sum of the static trip point, and then the correction factor, calculated on bootup, and the resulting static digital reference stored for comparison with the digitized transducer data?

I'm sure you agree that when the failure of a design could mean jail, or the poorhouse, the redundancy level for each catastrophic failure mode would be doubled or tripled, or pass the job. I'm not denigrating digital design, but I want to point that it has peculiarities to be considered.

Tom Callahan
tpcal@iu.net

Editor's note: A corrected version of Figure 1 from "Automotive Travel Computer" (INK 98, p. 67) is available as a downloadable PDF file on the Circuit Cellar web site at ftp://ftp.circuitcellar.com/CCINK/1998/Issue_98/correction.zip.

October Design Forum password:

Filter

INK ON-LINE

Your magazine enjoyment doesn't have to stop on the printed page. Visit *Circuit Cellar INK's* Design Forum each month for more great online technical columns and applications. Here are some of the great new on-line articles:

Columns

Silicon Update Online: Microchip on the March—
Tom Cantrell

Lessons from the Trenches: Get an Embedded Micro
and C Compiler off the Ground—George Martin

Forum Feature Articles

One More Wireless Trick to Stuff Up Your Sleeve—
Hank Wallace

Lost at C? Forth may be the Answer—Tom Napier

PIC Abstractions

Design Abstracts from our Design98 Contest
X-10 Temperature Sensor—Donald Blake
Compact Optical Image Scanner—John Luo
Heating Control System—Mark R. Wheeler

Missing the Circuit Cellar BBS?

Then don't forget to join the *Circuit Cellar INK* newsgroups! The cci newsserver is the engineer's place to be on-line for questions and advice on embedded control, announcements about the magazine, or to let us know your thoughts about *INK*. Just visit our home page for directions to become part of the BBS experience.

www.circuitcellar.com

NEW PRODUCT NEWS

Edited by Harv Weiner



GPS ANTENNA

Tri-M Systems' **Mighty-Mouse** GPS integrates a high-performance patch antenna and a state-of-the-art, low-noise, low-power consumption amplifier into a compact waterproof enclosure. Ideally suited for deployment as an external antenna for hand-held or vehicle-based GPS receivers, the antenna consumes less than 12 mA while producing 25 dB of signal gain.

An innovative universal adapter system enables Mighty-Mouse antennas to mate with GPS receivers from different manufacturers. Supported connectors include SMB, SMA, BNC, TNC, and MCX. The Mighty-Mouse comes standard with a 5-m RG174/U cable and features magnetic and permanent mounting. The unit measures 58 mm × 48 mm × 15 mm and weighs 65 g.

The GPS antenna is hermetically sealed and completely waterproof. It is manufactured with a die-cast metal baseplate, which acts as a ground plane, so you can mount the antenna anywhere without sacrificing signal-reception performance. A polycarbonate radome protects the antenna element, low-noise amplifier, band-pass filter, and buffer stage at operating temperatures from -30° to +85°C.

Tri-M's Mighty-Mouse antenna comes with everything needed to attach to the user's GPS receiver. Pricing starts at \$59 in OEM quantities, with a suggested list price \$79.

Tri-M Systems, Inc.
(800) 665-5600 • (604) 527-1100
Fax: (604) 527-1110
www.tri-m.com

REAL-TIME VIDEO IMAGE PROCESSING

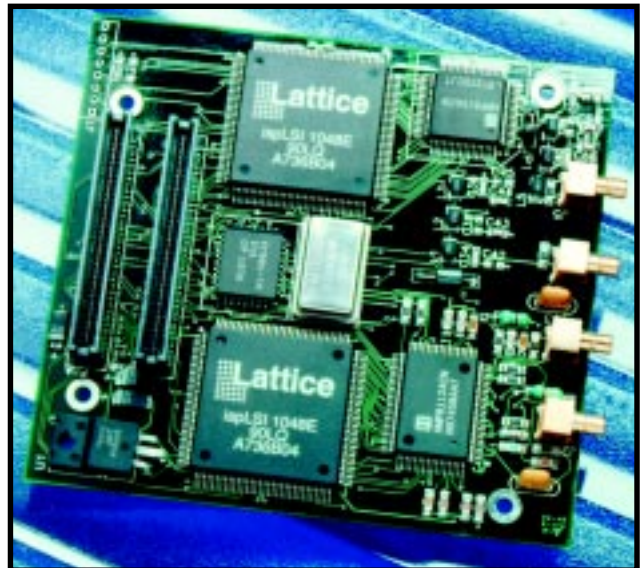
BittWare has announced a new video I/O mezzanine that interfaces NTSC/PAL video signals with SHARC processors in real time. The **bitsi-VIDEO** captures and displays composite and S-video signals, making it ideal for real-time image-processing applications requiring rapid scanning and peak I/O performance.

The bitsi-VIDEO mounts to SHARC-based host boards and uses a tightly integrated interface to transmit video signals to the processor on the host. Using square pixel or CCIR601 data formatting, the device encodes from, and decodes to, 8-bit YCrCb-format digital data. It delivers the signals to the SHARC processor on the host board for immediate processing or frame storage. Two link ports return processed video data to the bitsi-VIDEO for conversion back to analog video.

The bitsi I/O mezzanine standard is optimized to match the SHARC DSP's sophisticated I/O capabilities. A variety of off-the-shelf bitsi I/O mezzanines, including audio, control, video, and telephony interfaces, are available. The interface consists of 32 data bits, 26 address bits, 10 control signals, two DMA channels, four SHARC link ports, and three serial ports.

Pricing for the bitsi-VIDEO starts at \$1495.

BittWare
(800) 848-0436 • (603) 226-0404
Fax: (603) 226-6667
www.bittware.com



NEW PRODUCT NEWS



HARDWARE SIMULATOR

The **SIMICE** hardware simulator provides low-cost system debugging for the Microchip Technology PIC-12C5xx, '12CE5xx, and '16C5x eight-bit RISC microcontrollers. It works in conjunction with Microchip's MPLAB-SIM software simulator to provide non-real-time I/O port emulation.

SIMICE enables a developer to run simulator code for driving the target system. The target system can also provide input to the simulator code, enabling simple and interactive debugging. SIMICE features unlimited software breakpoints and PC communication via serial interface at speeds up to 57 kbps. It also supports source-level debugging.

The MPLAB Integrated Development Environment (IDE) gives users the flexibility to edit, compile, emulate, and program devices from a single-user interface. MPLAB software offers a project manager and program text editor, a user-configurable toolbar containing four predefined sets, and a status bar, which communicates editing and debugging information. A dynamic error capability creates rapid application development. MPLAB is available at no cost by downloading the software program from Microchip's Web site.

The complete SIMICE hardware-simulator system features a hardware I/O port emulator board, RS-232 cable, PICmicro target probe cables, and MPLAB IDE software. SIMICE is priced at **\$129**.

Microchip Technology, Inc.
(602) 786-7668
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

VERSATILE SINGLE-BOARD COMPUTER

The **SBC2000-074** single-board computer is a compact, low-power unit that features a Microchip Technology PIC16C74 running at 20 MHz. It is equipped with two RS-232 serial ports that run to 19.2 kbps, a real-time clock, watchdog timer, an 8-KB EEPROM, and five 8-bit A/D inputs. It also includes five programmable DIO lines, a low-power sleep mode, a PWM generator, an LCD port, and a keypad port. All this comes on a 2.8" × 1.3" board. Expansion of the SBC2000-074's hardware is facilitated by the board's VAST (Vesta Addressable Synchronous Transfer) network connector, which lets the user select from over 20 peripherals.

The SBC2000-074 includes Vesta Basic V.2, which is a Windows-based, IDE-driven compiled BASIC. The single-tasking software features floating-point calculations with 16-bit precision. The unit operates at temperatures between -40°C and +85°C and consumes 15 mA at 5 VDC. A low-dropout onboard voltage regulator creates optional direct connection to unregulated supplies ranging from 5.5 V to 24 VDC.

The SBC2000-074 sells for **\$64**.

Vesta Technology, Inc.
(303) 422-8088
Fax: (303) 422-9800
www.sbc2000.com



NEW PRODUCT NEWS

LCD MICROTERMINAL

The **ILM-216L** is an LCD module with a built-in serial interface. It works like a terminal, receiving text via RS-232 at 1200–9600 bps (8N1) and displaying it in large 5.9-mm (0.23") characters on a supertwist 2-line × 16-character LCD.

Formatting text for the display is a simple matter of using familiar terminal-control characters such as carriage returns, linefeeds, tabs, and so forth. Additional instructions control the backlight, position the cursor, define custom characters, and read four switch contacts. The ILM-216L automatically right-aligns selected

text, which is ideal for updating numeric fields with minimal programming overhead.

A unique feature of the ILM-216L is its nonvolatile configuration memory (EEPROM). Users can set the ILM-216L to perform any of the following

tasks at startup—display a custom splash screen, define custom characters, and turn on the backlight.

The ILM-216L draws only 5 mA with the backlight off and 40 mA with it on. The device measures 80 × 50 mm (3.15" × 1.97"), so it

fits into most standard 2 × 16 LCD mounting locations with a small overhang at the bottom.

Pricing for the ILM-216L ranges from **\$49** in single quantities to **\$29** in 100+ quantities.

Scott Edwards
Electronics, Inc.
(520) 459-4802
Fax: (520) 459-0623
www.seetron.com



NEW PRODUCT NEWS

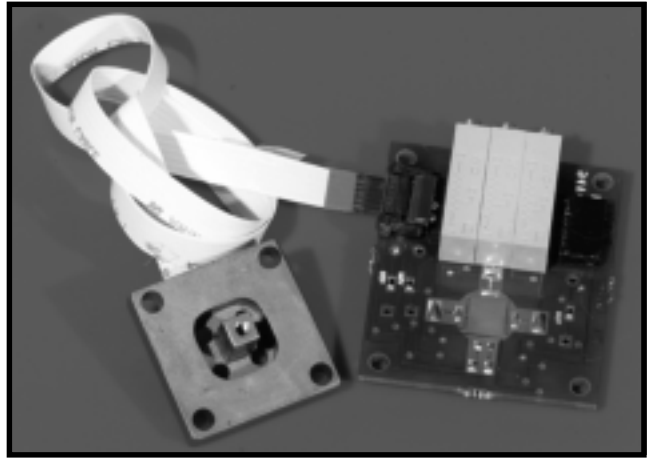
THICK-FILM SMART FORCE SENSOR

The **DX-300** is an all-metal, thick-film sensor designed for high-precision, multi-axis force sensing in harsh environments. Typical applications include coordinate measurement and robotics feedback controls, as well as precise multi-axis force measurement and fly-by-wire motion control.

The DX-300 has a network of conductive thick-film strain-sensitive elements (on a steel substrate) to measure microscopic strain. This network creates a force-measurement device that produces many times the output of even the best ceramic-based sensors.

Advantages of this construction include thermal stability, ESD protective design, high output, rotational accuracy, chemical stability with different cap chemistries, and no crack-propagation problems. So, the sensor circuitry will never fatigue or change output with time. The DX-300 features noncompliant palm- or finger-activated motion and force-sensing capabilities in a high-output, precise smart-sensor design.

The sensors are offered in two smart-sensor systems—the cursor control Integrated Block system that provides an RS-232 or a PS/2 sensor output for precise cursor control, and the force measurement/motion



control package that provides a 0–5-V output in all three distinct axes of input.

The DX-300 costs **\$165** in quantities of 500, and it is available with full signal-conditioning electronics for a slightly higher price.

Bokam Engineering
(714) 513-2200 • Fax: (714) 513-2204
www.bokam.com

FEATURES

12

X-Y Graphing Data
Logger

24

Time-Domain Filter
Simulations Using C++

30

Breaking Nyquist

36

Digital Frequency
Synthesis

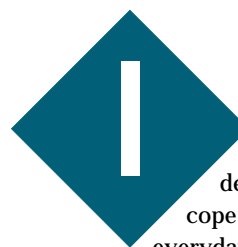
FEATURE ARTICLE



Alberto Ricci Bitti

X-Y Graphing Data Logger

With more data than he can handle (and always in some inconvenient place), Alberto constructed a powerful, handheld, programmable data logger from his Casio pocket calculator. And as a reward, Design98 judges made it their “first PIC.”



Like any other designer, I have to cope with lots of data everyday. Raw data in need of analysis comes from every design phase and from all related sites. From writing specifications to development, from production tests to on-site verification, we end up with tons of measurements.

A graph is often the best way to point out the key features of what you measure. It's useful for instantaneous communications and easy to document for later reference. It's accepted for corporate quality system records, too.

PCs are powerful graphing tools, and maybe that's why almost all recent instruments have some kind of PC interface. So, you just take out your dazzling new computer-interfaceable meter, connect it to nearest PC, and start measuring. Right?

Wrong. Sometimes you want to take measurements in the field, and you can't take the instruments out of the lab. Something other than the PC can collect the data, but it's fooled by grounding problems.

Other times you need a battery-operated instrument, but a laptop is

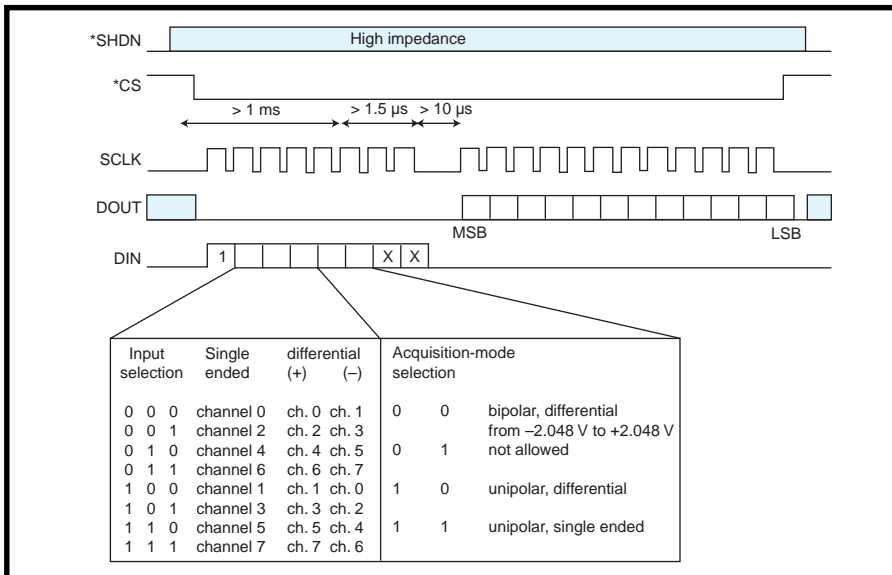


Figure 3—The analog inputs of the MAX186 can be switched to bipolar and pseudodifferential mode by changing the control word. In bipolar mode, the input spans between ± 2.048 V instead of the usual 0–4.096 V. The last two bits in the control word select the power-down mode. Here, they are overridden, forcing the shutdown pin (*SHDN) low.

the calculator math) to go from ideas to prototype. This technique also simplifies the problem-solving path: a simple concept, simple electronics, and simple software yield a complex result.

You also get lower power consumption (at least 200 h on standard batteries), which is more than 20 times a comparable PC-based solution.

Also, using the calculator cuts cost. Just one, off quantity, sells for the price of an LCD alone. Overall costs are more than 20 times less than any PC-based solution of comparable power.

You get increased flexibility, low cost, and an easily replaceable flash-reprogrammable input board, enabling you to retain the calculator. Plus, only data acquisition and transmission need to be tested, and you save on production costs because you don't need complicated plastics and electronics.

Documentation is simpler, too. Just add a "Capturing data" chapter to the calculator's manual.

Size and weight are also reduced. Even prototypes are handheld, weighing only 290 g (including batteries), and series production sizes can be reduced further.

LOW-POWER GUYS

Every successful project relies on component selection, especially battery-operated designs. Maxim's MAX186 is a low-power 12-bit ADC. It needs

minimal external hardware and features a nice internal 4.096-V reference that sizes each step to a handy 1 mV.

The MAX186 sports eight single-ended inputs, which is more than enough for most applications. The same inputs can be reconfigured as four bipolar and pseudodifferential inputs by simply changing the command word. It's suitable for reading data directly from a large variety of sensors (see Figure 1).

The MAX186 generates its clock internally, and the entire operation is controlled through a four-wire SPI, QSPI, or Microwire serial interface. The device is put in standby mode via a three-level input pin or with a software command word.

It consumes 1.5 mA typical while operating, which drops to a mere 2 μ A in full power-down mode. There is also an interesting fast power-down mode (not used in this design) that consumes 30 μ A with a wake-up time as short as 5 μ s.

If 10 bits are enough for your taste, you can replace the MAX186 with the MAX192. It's fully pin- and software-compatible with the MAX186. Price aside, the only difference resides in the precision of the two lower bits, which aren't guaranteed for the MAX192.

The chip needs a single 5-V power supply. With such a low power requirement, the drain of the power regulator

itself becomes important. Ordinary regulators such as the 78L05 can easily drain more current than the whole circuit.

You need specific low-dropout regulators like the LM2936 to get maximum battery life. The LM2936 has a quiescent current of only 9 μ A, and it's internally protected from reverse battery connection.

FLASH RISC GLUE

An eight-bit RISC microcontroller glues the ADC to the Casio serial input. Because my goals included rapid development time and low cost, I focused on small flash- or EEPROM-based RISC microcontrollers.

The simplified RISC architecture is easy to learn, and erasable parts let you concentrate on the problem instead of the UV eraser. As a bonus, these parts usually have simple, ultra-cheap PC-port-based programmers.

I chose the PIC16C84 (an EEPROM part) and the newer PIC16F84 (an improved flash version) from Microchip. A useful characteristic of the PIC architecture is its support of data tables that are as long as the program memory. This characteristic is a result of the RETLW instruction.

These PICs are powerful enough to handle serial communications entirely in software with a 4-MHz clock. A faster part (you can find 50-MHz PIC clones) is unnecessary.

The PIC draws only a few microamps when sleeping (even with the watchdog timer enabled), and best of all, it's cheap and available. MPLAB, a professional grade assembler and simulator, is distributed for free by Microchip, along with lots of useful libraries.

THE GRAPHIC ENGINE

The Casio FX-9750G graphing calculator has 32 KB of RAM for data or programs and a 64 \times 128 black and white LCD. The FX-9750G is a member of a larger family that includes models with color LCDs and up to 64 KB of memory. It's a powerful and enjoyable math tool, but it costs about the same as a graphic LCD module alone.

It runs on four LR03 batteries for 200 h, and sensitive data and programs are maintained for up to one year by a

Photo 1—The circuit fits inside a small plastic box just as large as the calculator. The micro jack connector hangs out, connecting to the calculator serial port. The box is stuck to the bottom of the calculator for operating. This arrangement gives the calculator a stable and comfortable slope.



separate lithium battery. I'm glad the Casio folks elected to use a standard miniature stereo jack as the serial port connector for external peripherals.

The data protocol is built around a standard 9600-bps half-duplex serial stream, with one start bit, two stop bits, and eight data bits with no parity.

Interfacing the FX-9750G to a PC is a matter of adapting TTL-to-RS-232 levels. A MAX232 can do the job. I built one of the dozens of similar circuits I found on the Internet, coupled with the FA-122 Windows backup software.

I haven't tested the compatibility with other calculators (besides the FX9750G), but I don't expect any differences in the protocol between similar models.

CASIO PROTOCOL

The Internet is now an invaluable resource for designers. After a night of browsing, I found some commented programs for loading calculator pro-

grams to and from a PC. I found only partial information about how to transmit or receive single variables instead of programs, but it served as a good starting point.

I assumed that the variable transfer format probably wasn't so different from the program transfer format. Notably, even the official Casio FA-122 backup software can be found online.

I knew that the communication was a 9600-bps half-duplex TTL serial stream, and I knew communications should start with ACK/NACK-style, single-character messages. Packets follow, starting with colons (\$3A) and terminating with checksums.

I also had PIC samples, some good ready-made serial and BCD routines,

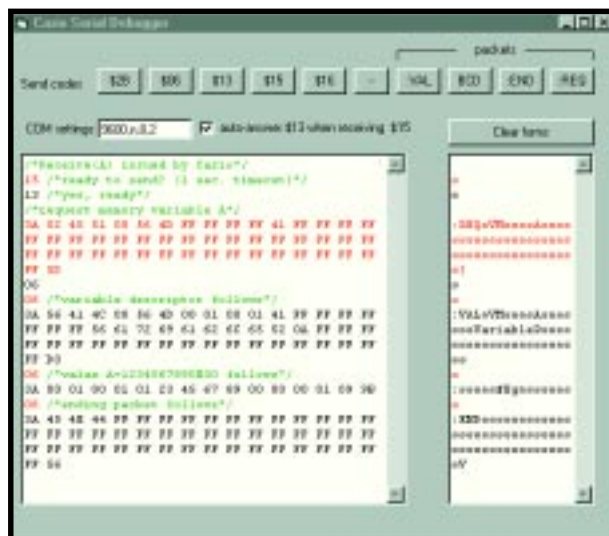
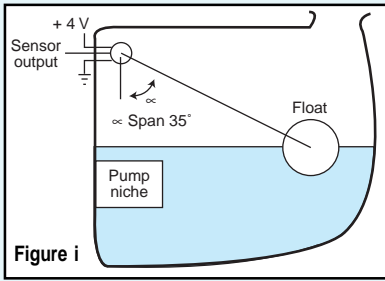


Photo 2—I used Visual Basic to discover the inner secrets of the Casio protocol. With VB5 it's easy to set up a quick and dirty serial protocol debugger. Tasks responding automatically to a control character, trying various serial speeds, or sending packet strings are left to a simple program loop. Thanks to Visual Basic's interpretative nature, you can stop the program, issue any other command manually, then continue execution. This way you avoid writing lots of code. Here, the characters sent by the Casio are in red, and the analog interface response is in black.



Straightening Curves—Linearizing a Tank-Level Sensor

Most sensors' outputs are nonlinear. To use them practically, they must be straightened using interpolation tables or analytical models.

Regression models are popular analytical tools for substituting data tables with clean, continuous functions that are elegant and practical. During calibration, function substitutes require only three or four points to be measured, as compared to the many more points that tables require.

The volume of liquid in the 25-l tank in Figure i is measured using a float and potentiometer. The output is proportional to the angle. The tank's irregular shape makes an analytical approach impractical. It's better to measure the sensor output at known liquid quantities.

Photo i—The output is connected to the x input, and the program in Listing 1 is run. Each cycle, exactly 1 l of liquid is added until the tank is full. At the end, List1 stores the liters of liquid, and List2 stores the sensor's output level. Successive data manipulation is done manually.

Photo ii—Selecting STAT gives you the sampled data in its raw form, as it appears on the default entry page. Here, editing and sorting can take place. GPH1 lets you view data graphically, and SET lets you check the current graphing preferences.

Photo iii—The preferences for GPH1 are set as a scatter graph with small box markers. I want to plot the liters over the sensor's output, so I select List2 for x and List1 for y . Showing the inverse function is simply a matter of exchanging coordinates here. Exit goes to the previous menu.

Photo iv—The display scales automatically to fit all the data, but you can zoom in or pan in each direction. The dotted grid is shown every 50 mV for x and 5 l for y . Note the gap at about 5 l (after the fifth marker), which is due to the pump niche.

Photo v—To draw the first-order regression line, just press X . Other regression models (e.g., median-median, second- to fourth-order polynomials, logarithmic, and exponential) are obtained the same way.

Photo vi—The first-order regression is too coarse for reliable measurements, but it's easy to try other curves. The second-order one, shown here, fits the data nicely.

Photo vii—Zooming shows that the error at the pump niche gap is negligible.

Photo viii—Pressing X^2 brings up the coefficients and the regression formula, ready to be used for linearizing the sensor's output. Thanks to the second-order characteristic, a full data table is no longer necessary. I can do the calibration with only three points.



and a microjack plug. That's all I needed to start experimenting.

Using a TTL-to-RS-232 level converter, I connected the calculator to the PC serial port so I could monitor what's going on.

After I issued the `Receive(X)` command on the Casio, it sent out a \$15 character and, after 1 s, it aborted (a \$22 character sent to the PC prior to interrupt communications).

I made a Visual Basic program to issue various characters after that attention request (see Photo 2). I quickly discovered that the calculator was waiting for a \$13 character.

A request packet from the calculator follows this vital sign from the PC (50 bytes, starting with `:REQ`), where the calculator defines the variable to be sent. As expected, all nontrivial packets start with a colon character and are terminated by a simple negated checksum. With the exception of the last packet, all require an acknowledgment message from the recipient (`$06`).

Since I didn't know how a value packet is made, I reversed the situation, making the Visual Basic program act as a Casio calculator issuing the `Receive(X)` command that was just received. At the same time, the real Casio calculator issued `Send(X)`. In this reversed setup, the calculator

issued a 50-byte variable-address packet, starting with `:VAR`.

I reiterated this process, continuously reversing the sender with the receiver, sometimes simulating `Send(X)` and sometimes simulating `Receive(X)`. Packet after packet, the whole protocol was discovered, as depicted in Figure 2.

MAIN CODE

The software copes with communication protocols over a serial line, combining sleep mode and watchdog techniques to achieve minimal power consumption, low-power ADC driving, and data conversion to the Casio format.

Complete packet templates are stored in data tables in program memory. At this point, I found `RETLW` invaluable.

Only segments with variable values are replaced by real data read from the ADC in real time. Most parts of the input packet are ignored, and only the variable name is stored to select the right ADC input. When issuing `Receive(X)`, you can specify any other variable name to select a different input.

Even the simpler communication protocol must deal with errors and interruptions. To keep program overhead low, I set up the watchdog timer to reset the device if it waits for an answer from the calculator for more than 2 s.

Once reset (and for most of the time), the device is left in sleep mode. It's awakened by another WDT timeout or by receiving a character.

In the former case, the device goes back into sleep mode. In the latter case, the LED flashes, the ADC awakens, and the communication flow restarts.

When the micro sleeps, the ADC is left in full power-down mode. The shutdown pin is a three-level selection input, and the MAX186 sleeps with that pin at 0.

It is awakened by putting the pin in a high impedance state, which is done long before the conversion starts to let the voltage reference capacitor charge completely.

The MAX186 input can be reconfigured to bipolar input mode (with input spanning from $-V_{ref}/2$ to $+V_{ref}/2$) or pseudodifferential input mode by changing the command word that is ORed in the `ReadADC` code segment. Figure 3 shows the signals involved and how the control word is made.

The Casio needs variables in BCD format with separated exponent digits. The most significant digits come first. The BCD conversion and the bit-banging serial-port routines are derived from public-domain Microchip libraries.

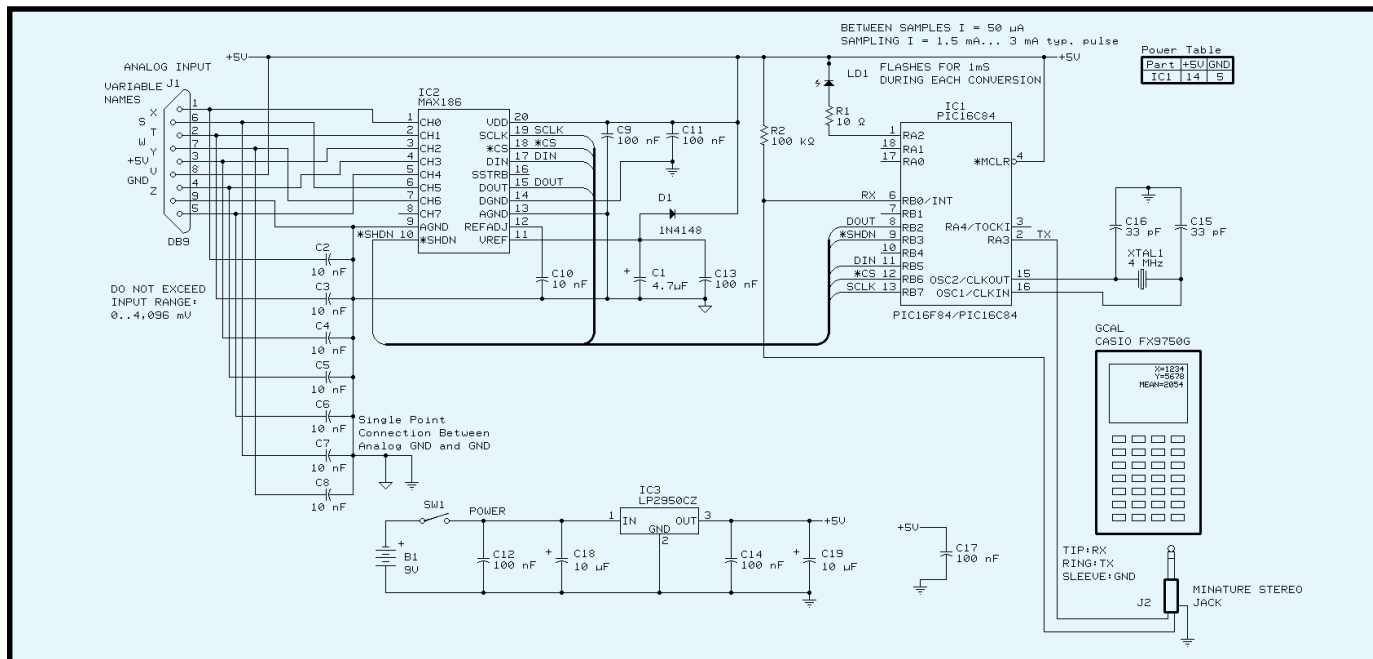


Figure 4—Using a graphing calculator as the output and processing device results in a simplified circuit diagram. The 5-V power rail is brought to the input connector to feed external sensors. If the quiescent current does not concern you, then the popular 78L05 can replace the LP2950CZ.

INSIDE THE BOX

The final circuit diagram is shown in Figure 4. Since this is a low-power, unshielded, mixed A/D circuit, the overall result depends on the quality of the layout.

Even though we're all accustomed to 12-bit converters, you must take great care to correct grounding so you have stable readings of the least significant bits. The analog and digital grounds must be kept separate. They have to be joined only at one point (as near as possible to the regulator ground). Bypass capacitors are mandatory, and capacitors on the inputs are equally necessary.

The MAX186 has a good pin layout, which helps separate input lines from data lines, and a stable readout is easily obtainable. But in noisy environments, consider averaging in software to further reduce uncertainty.

The eighth analog input is left unconnected on the prototype. This arrangement enabled me to bring the power out to the nine-pin input connector, thus powering external sensors.

However, it's fully supported by software, so if you need it, simply add an input capacitor and feel free to use it.

Of course, power supply is critical in every battery-operated device. Power-up rise times must be short because there is no external reset circuitry for the micro. The MCLR pin is tied directly to V_{CC} .

The low-power regulator is more delicate than a regular 78L05. It requires an output electrolytic capacitor—better if it's tantalum.

The required current is impulsive, consisting of a 50- μ A offset (sleeping micro and ADC), 1.5 mA during serial communication with the calculator, a single pulse of 1.5 mA for a few milliseconds (A/D conversion), and a 1-ms peak for LED flashing (current given by the LED series resistor).

The LED flashes once per conversion. I recommend a red LED because it produces more light with the same current.

I brought the power supply to input pin 8 in case some sensor needs it. Be sure to take sensor power requirements

into account when estimating battery life. If the sensor sinks a significant amount of current, consider powering it through one of the free PIC I/O pins, which will power off when not in use.

I assembled the whole circuit using a prototyping board, and it fits inside a small plastic box that's only the size of the calculator. The box contains all the circuits and the battery. Only the micro jack connector (connecting to the calculator serial port) hangs out.

As you see from Photo 1, there's a lot of space left on the board. If possible, use a 90° jack to keep the unit even more compact and rugged.

SETTING UP THE CASIO

While operating, the analog interface box is stuck to the bottom of the calculator, below the LCD, with TESA removable biadhesive strips. This arrangement gives the calculator a stable and comfortable slope, and it leaves the input connector in a handy place, free from obstacles.

To reveal the micro jack socket, remove the rubber cover that comes

with the calculator. If the jack is left unconnected or the unit is powered off, a Com error message is displayed.

You don't need any particular programming skill to use the acquisition unit. One instruction does it all.

Issuing `Receive(X)` directly (Recv softkey in the PRGM I/O menu) displays the value read from channel *x*. You can manipulate the *x* variable like any other ordinary variable, exactly the same way you would if you entered it manually. Value is expressed in millivolts, and ranges from 0 to 4095.

`Receive(X)` is usually issued under program control, as in Listing 1. But, I like to manipulate graphs manually. It's an instructive, highly interactive way.

Nevertheless, every keystroke can be replaced by a matching keyword to be issued under program control. From the programming standpoint, complex tasks (e.g., displaying a whole graph or computing a fourth-order regression) count as only a single instruction.

You can specify other variable names instead of *x*. Each variable

selects a different input. Variables supported are *x*, *y*, *v*, *w*, *z*, *s*, *t*, and *u*. Other names are seen as aliases and won't cause errors.

FIRST GRAPH

Look at the real-world example in the sidebar "Straightening curves—Linearizing a Tank-Level Sensor." Here, a float drives a potentiometer sensing the liquid level in a 25-l, irregularly shaped tank.

I want to figure out the relationship between the sensor output and the quantity of liquid left in the tank and to gather enough data to build a model for the control processor.

The complex relationship between angle, height, and volume makes an analytical approach impractical. It would be better to measure the sensor output at known liquid quantities.

The potentiometer output is brought to input *x*. It's powered at 4 V to avoid damaging the inputs.

In Listing 1, you see the simple control program that's required. `List` is the equivalent of an array in the

Casio world. `Seq` allocates the memory space for a list and initializes it. Here, two lists—one for the liquid quantity and one for the sensor output—are created and filled with data in a simple for-next loop.

At each cycle, 1 l of liquid is added and a measurement is taken. This process continues until the tank is full. When the program ends, the two lists hold the quantity of liquid as well as the sensor's output level.

Even if data visualization commands could be included in the program, it is convenient to look at nonrepetitive tasks manually.

The STAT menu lets you examine the data tables. If you select the GRPH submenu and GPH1, Graph1 is then displayed as an *x-y* scatter graph of `List1` (liters) over `List2` (sensor output).

Although nonlinear, it's immediately clear that the sensor output is suitable for measurements. An anomaly at about 5 l, due to the presence of a pump niche that reduces the available volume, is equally evident.

MODELING DATA

By means of regression, the measured data can be shaped into a function that fits into a nice formula. Formulas are not only easy to implement, they also give us a better understanding of the data and can significantly reduce the number of points needed for calibration.

In this example, the original data set is made of 26 points. Knowing that a sensor's output is a second-order function enables a three-point-only calibration without appreciable degradation. The calibration procedure can then be reduced to reading the output at three convenient positions (empty, full, and halfway).

On the Casio, complex regressions are a single keystroke away. When a graph is displayed, as in Photos v, vi, and vii in the sidebar, the softkey line lists a variety of popular regression functions to choose from. Pressing x brings out the coefficients for first-order regression line. DRAW puts a graph over the sampled data.

As you see from Photo vi, a simple straight line leaves a lot to be desired. The shift-zoom combination brings up the zoom menu, while pressing one of the arrow keys pans the whole display in the indicated direction.

The X^2 softkey switches to the second-order regression. This time, the graph is very near to almost all the samples. The overall result doesn't vary appreciably regardless of the regression order (e.g., x^3 , x^4). Other regression models (e.g., logarithmic, exponential) don't give significantly better results.

You can interact with your data and try out as many functions as you like, exploring the possibilities without having to write a single line of code. It is impossible to list here all the functions in the 425-page Casio FX9750G user guide. There's also a full set of statistical tools—useful when monitoring production sample parameters, weather data, pollution, and so on.

A BROADER VISION

Since this design was announced in INK 95, I've received lots of E-mail from interested readers. Each one had a different vision of what this little design can be used for—from tracking the accuracy of a GPS-locked PLL and

Listing 1—This simple program makes 25 samples and places them in *List2*. Another list is filled with ascending numbers used as x-axis values on an x-y graph. *List1* is the equivalent of an array in the Casio world.

```
Seq(N,N,0,25,1) --> List1
Seq(0,N,0,25,1) --> List2
For 1--> N To 26
  Receive(X)
  X --> List2[N]
  Locate 1,1, N
  Locate 10,1, X
While (Getkey): WhileEnd
Next
```

plotting weather data, to showing a waveform from an instrument and monitoring a central heater's operation, or counting people coming in a door.

The number of possible applications exploded. Built-in display and the capability of being programmed, combined with very low cost, are the key factors.

I encourage you to expand the capabilities of the data logger. The code is fully commented, and the PIC program memory is only half full.

I do have a couple suggestions. As a first step, try adding a pulse-counter mode. There are lots of things worth counting, and many sensors (e.g., Steve and Jeff's lightning sensor [INK 90]) have pulse outputs.

Secondly, drive outputs by implementing the Casio command Send(X) to make the interface bidirectional. The Casio is slow but powerful enough to read in the input, make some computations, and send output to the outside world. In this way, it would migrate from the world of monitoring and displaying events to the broader world of (pocket) computer control. ☱

Alberto Ricci Bitti is a software designer at Eptar, an industrial-controller firm. He has written software for systems such as meteorological equipment, specialized TV sets, professional satellite devices, industrial machinery controllers, and energy-management devices. You may reach Alberto at a.riccibitti@ra.nettuno.it.

SOFTWARE

Source code in official Microchip mnemonics is available via the Circuit Cellar web site.

REFERENCES

- MPLAB IDE free assembler and simulator software, 1997 technical library, www.microchip.com.
- Casio, FA-122 software and PC link schematics, members.tripod.com/~carolino/.
- Casio, *FX9750G User Guide*, A340606-27.
- Maxim Integrated Products, *MAX186-MAX188 low-power, 8-channel, serial 12-bit ADC*, Datasheet 19-0123, August, 1996.
- Maxim Integrated Products, *MAX192 low-power, 8-channel, serial 10-bit ADC*, Datasheet 190123, March, 1994.
- Microchip Technology, *PICmicro mid-range MCU family reference manual*, Datasheet DS33023A, December 1997.

SOURCES

PIC16C84, PIC16F84
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 786-7277
www.microchip.com

MAX186
Maxim Integrated Products
(408) 737-7600
www.maxim-ic.com

FX9750G
Casio Electronics Co., Ltd.
www.casio-usa.com

LM2936, MAX186, PIC16C84, PIC16F84
Digi-Key Corp.
(218) 681-6674
Fax: (218) 681-3380
www.digi-key.com

FEATURE ARTICLE

Glenn Parker

Time-Domain Filter Simulations Using C++

When you have to work with filters—and what hardware engineer doesn't at some point?—you'll want to try out Glenn's complete C++ FILTER class. He's ready to prove how easy it is to integrate this class into embedded apps.



Early every hardware engineer (analog or otherwise) deals with frequency selective circuits, or filters, at some point in his or her career. Filters are everywhere, and most electronic systems can't function without them.

Circuit simulation is fairly straightforward today, especially with readily available simulation software. But, some engineers still write software (for speed or to handle aspects of their application not addressed by commercial software).

This article presents a complete C++ class for simulating electronic filters. If you need a refresher course, check out the sidebar "Analog Filter Basics."

The source code is implemented for analog filtering, but you can easily adapt it to handle digital filters or any analog system that can be characterized by a transfer function in the frequency domain. A Windows executable complete with source code is also available which uses the class to simulate various analog filters, including elliptic types.

Digital or discrete-time filters are implemented in software or through dedicated DSP hardware. Analog or continuous-time filters are generated by designing a network whose input impedance changes with frequency to realize a desired response curve.

For this reason, analog-filter specifications are typically given in the frequency domain. Still, it's important to consider the time-domain characteristics of analog filters whenever waveform distortion (e.g., in digital demodulation systems) or signal delay is critical.

TRANSFER FUNCTIONS

Analog-filter designers are undoubtedly familiar with how the Laplace Transform relates continuous-time with the frequency domain. There is a parallel relation between discrete time and frequency via the z -transform.

The coefficients of a function in the z -domain can be used to construct a constant-coefficient difference equation, which provides a direct method for time-domain simulation:

$$y[n] = 2x[n] + x[n - 1] - 3y[n - 1]$$

where $y[n]$ is the output at time Δtn , and $x[n]$ is the input at time Δtn . By iterating n , this system's time-domain response to a changing input can be calculated. The constant Δt is simply the time interval between $n - 1$ and n .

Once the difference equation coefficients are found, a real-world signal can be sampled with an ADC and filtered by hardware with dedicated DSP devices or microcontrollers or by some other method implementing the difference equation. The same idea lies behind digital filters as well.

The difference equation coefficients can be tweaked to realize nearly any magnitude and phase requirement. Or, you can take the coefficients and funnel them through a simulation engine like the one presented here to see what the original analog filter looks like in the time domain.

The only missing link is how the s -domain function gets into the z -domain. There are several acceptable methods for accomplishing this, each with benefits and pitfalls. The class I present here uses the bilinear transform.

The nonlinear mapping inherent in this transform compresses the frequency response. The critical filter frequencies can be moved slightly to compensate for the magnitude response compression, but in most cases, the phase response is noticeably distorted.

Figure 1 shows a linear-phase filter response before and after the bilinear transformation. Because of its inevitable phase distortion, this transform is not suitable for modeling linear-phase or flat group-delay filters.

Predistortion or prewarping is often applied to the original filter so the transformed filter's magnitude response looks like the original. Prewarping is done by shifting the s -domain cutoff frequency before transformation to force the transformed filter's cutoff to occur at the correct frequency.

The new cutoff frequency is given by:

$$\omega = 2f_{\text{sample}} \times \tan\left(\frac{1}{2} \times \frac{\Omega}{f_{\text{sample}}}\right)$$

where Ω is the discrete-time frequency and ω is the corresponding continuous-time frequency.

QUANTIZATION ERRORS

Although a single section can be used to realize any filter order, coefficient quantization is an important consideration. In a single-section filter, each pole or zero contributes significantly to the coefficients of the difference equation.

If one pole or zero experiences severe round-off problems, the entire filter is affected. This situation is most easily overcome by cascading low-order sections to realize high-order filters.

Since each pole or zero only contributes to the coefficients in a single section, quantizing a high-Q pole near the $j\omega$ axis only has an effect on one section instead of the whole filter.

While it's true that any section in a cascade has an effect on the overall system, the error contribution is much smaller than in a single-section filter realization.

By cascading smaller sections (no more than two or four poles per section), the simulator can handle larger filter orders, but that's at the expense of speed.

The solution: use small sections and eliminate as many calculations per iteration as possible. The class I discuss handles huge filter orders, and in benchmark tests, it was at least 100 times faster than commercial simulators.

Filter orders as high as 100 have been simulated using this example application. Obviously you wouldn't ever build such a filter, but it's useful for simulating a brick-wall effect.

CALCULATION MODES

The class as presented is designed for use in a time-domain simulation engine. It operates in two calculation modes—precalculation and time-step iteration.

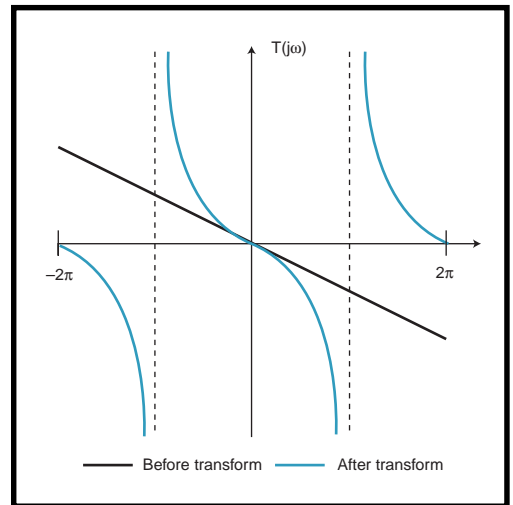


Figure 1—This graph shows the phase response of a linear phase filter before and after applying the bilinear transform.

The precalculation mode is used in cases where the input signal has been precalculated and placed in an array, (e.g., my sample application, where data is fed through a filter only).

The second mode is used when iteration is needed, and the input value is only available in time steps. For example, most feedback systems have inputs that depend on outputs, so it's not possible to know what the filter's next input is until you know the current output.

If this all seems confusing, don't worry. You can use the sample application to simulate filters without ever digesting the source code.

USING THE FILTER CLASS

The three most-used functions in the FILTER class are shown in Listing 1. The first two constructor arguments are set equal to #define constants given in the source file. They determine the filter type (e.g., low pass) and shape (e.g., Chebyshev). Table 1 defines the remaining arguments.

As you might guess, calling either TransferFunc() or TimeStep() defines the simulation mode you're operating in. TransferFunc() takes a pointer to the input and output data and the number of points in those arrays. TimeStep() takes a pointer to the input and output data and the current time step.

Additional useful functions (e.g., GetEllipticRoots()) are given in the sample application, but only the

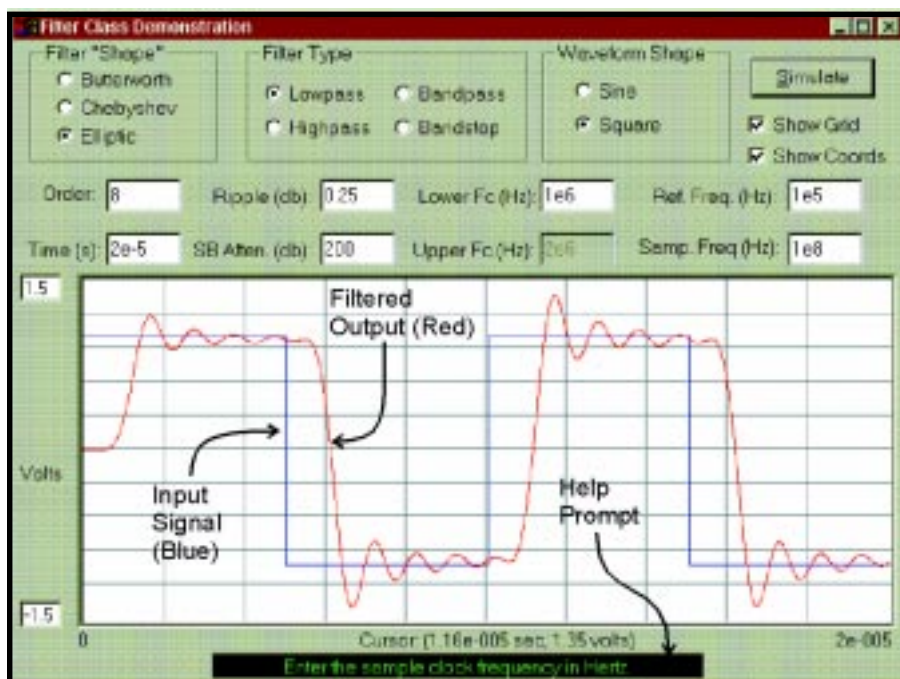


Photo 1—This screen capture of the sample application shows a simulated response.

Analog Filter Basics

According to a 1984 edition of *Webster's Dictionary*, an electronic filter is “a device that rejects certain signals while passing others.” An analog filter is a circuit containing reactive elements and whose transfer characteristics vary with frequency.

Usually, a certain band of frequencies is allowed to pass, while other frequencies are attenuated or stopped altogether. Figure i shows a typical frequency response, along with an illustration of several common filter terms.

The frequency (or frequencies) where the pass band ends is called the cut-off frequency. There are four common filter types:

- low-pass filters pass frequencies from DC to the desired cut-off frequency. All other frequencies are attenuated.
- high-pass filters attenuate frequencies from DC to the desired cutoff and pass higher frequencies.

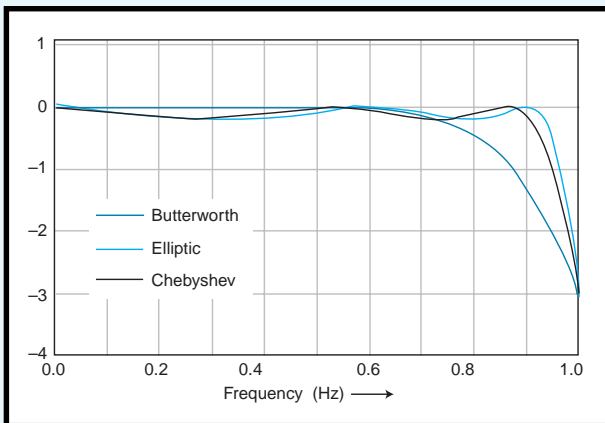


Figure ii—Here's the pass-band response of three popular filter types. These filters have been designed for -3-dB cutoff at 1 Hz.

- band-pass filters pass a band of frequencies between the two desired cut-off frequencies. All frequencies below and above the pass band are attenuated. The response shown in Figure i is from a band-pass filter.
- band-stop or band-reject filters stop a band of frequencies between the cut-off frequencies. All frequencies below and above the stop band are passed.

Ideally, the stop band would border the pass band exactly, and the transition band would have zero width. However, in real filters, the transition band always exists, and there are many factors involved that control the response shape.

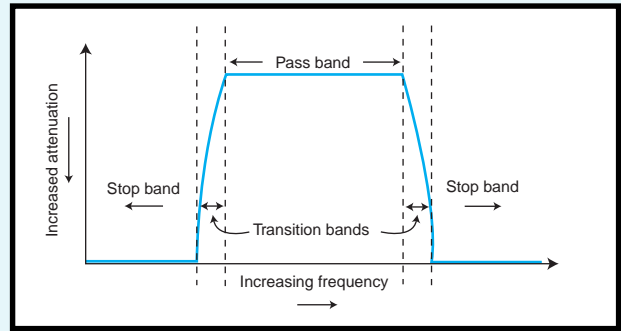


Figure i—The pass-band response shown here illustrates common filter terms.

A filter's transfer function is defined as:

$$T(s) = \frac{V_{out}(s)}{V_{in}(s)}$$

where s is the Laplace Transform complex frequency variable, V_{out} is the filter's output voltage, and V_{in} is the Laplace Transform of the signal being input to the filter. The magnitude response shown in Figure i is found by substituting

$$s = j \times \omega$$

and taking the transfer function magnitude where

$$j = \sqrt{-1}$$

and ω is 2π times the frequency of interest in hertz.

A filter's order is equal to the degree of the denominator polynomial in the transfer function. This value is directly related to the complexity of the filter necessary to realize the transfer function. In

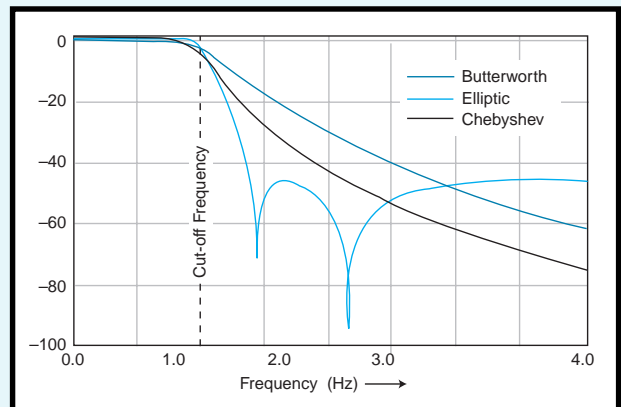


Figure iii—This graph shows a stop-band comparison of the filters used in Figure ii.

(Continued on page 28)

order	Filter order (number of low-pass poles)
ripple	Pass-band ripple in dB. Used only for Chebyshev and elliptic filters.
stopBandAtten	Minimum stop-band attenuation in dB. Used only for elliptic filters.
fcl	Lower cut-off frequency in hertz
fcu	Upper cut-off frequency in hertz. Used only for band-pass and band-stop filters.
sampleFreq	Sample frequency in hertz. This value should be several times greater than the largest frequency specified above to avoid aliasing.
gain	Maximum pass-band gain. This parameter is linear and specified in V/V.

Table 1—Here are the definitions of the *FILTER*-class constructor arguments.

necessary functions for getting started are listed here. However, the *FILTER*-class source code is heavily commented and should be easy to follow.

A WINDOWS APPLICATION

My Windows application uses the *FILTER* class to simulate a variety of analog-filter designs. Photo 1 shows a sample screenshot from the program.

An eighth-order elliptic low-pass filter with a cut-off frequency at 1 MHz was simulated. The pass-band ripple is set at 0.25 dB, and the minimum stop-band attenuation is set at 200 dB. The reference (input) signal is a square wave at 100 kHz, and the sampling frequency is 100 MHz.

The help prompt changes each time the input focus moves to a new control. I tried to make the prompts self-explanatory, but given onscreen space restrictions, they may appear cryptic. I hope the help prompt will clear up any questions. If you don't understand a prompt, you can change the number repeatedly and hit Simulate until it's clear.

The only prompts that don't require resimulating to take effect are the vertical range inputs. The calculated response is redrawn with the new

range anytime one of the range prompts loses focus and the value changes.

ARBITRARY SYSTEM SIMULATION

For each section, the *FILTER* class constructor calls the function:

```
void CalcSection-
Coeff(FILTER_SECTION
*s, complex *pole,
complex *zero, int
filterType, double
wc1, double wc2)
```

This function calculates the *a* and *b* coefficients used to calculate the cascaded *FILTER_SECTION* responses.

Instead of using the included functions for calculating poles and zeros, an array of poles and zeros could be passed to the *FILTER* constructor for use in building the *FILTER_SECTION* cascade. These poles and zeros should be normalized, since *CalcSection-Coeff()* scales the roots by the two cutoff frequencies.

ADAPTING THE CLASS

Listing 2 shows a portion of the *FILTER* and *FILTER_SECTION* classes. The *FILTER* constructor takes analog-filter parameters and fills in each *FILTER_SECTION* accordingly. You could say it designs digital filter coefficients that realize the analog filter.

The *FILTER* constructor takes analog filter parameters and calculates the *a* and *b* coefficients via the bilinear transform. These coefficients are the same ones used in a hardware implementation of an Infinite Impulse Response (IIR) filter.

You can create a complete IIR-filter simulation engine by adding over-

(Continued from page 27)

other words, higher order filters require more parts to build. In general, the higher a filter's order, the more narrow the transition region, and the more square the response shape.

There are three popular magnitude response filter types available. Butterworth filters have a flat response in the pass band and monotonically increasing attenuation into the stop band.

Chebyshev filters have a smaller transition band than Butterworth for the same order and better stop-band rejection. However, this is at the expense of ripple inside the pass band.

Elliptic filters have a more narrow transition band than Chebyshev for the same order, and better stop-band rejection by including zeros of transmission in the stopband. However, elliptic filters require more parts to build than Butterworth and Chebyshev filters.

These three response types are compared in Figures ii and iii. The filters are all fifth order and have 3 dB of attenuation at the cutoff frequency (1 Hz). The elliptic and Chebyshev filters were designed with about 0.43 dB of pass-band ripple. The elliptic filter was designed to maintain at least 45 dB of attenuation in the stop band. More attenuation could have been requested, at the expense of widening the transition band.

Chebyshev and Butterworth filters are relatively easy to design and can be built from the same topology by simply changing component values. Elliptic filters require more complex structures to realize the stop-band transmission zeros and can drastically increase both part count and component tolerance sensitivity.

Listing 1—These are the three most used functions in the *FILTER* class.

```
FILTER::FILTER(int filterType, int filterShape, int order,
double ripple, double stopbandAtten, double fcl, double fcu,
double sampleFreq, double gain);
void FILTER::TransferFunc(double *input, double *output, int
numPoints);
void FILTER::TimeStep(double *in, double *out, int timeStep);
```

Listing 2—This source-code snippet illustrates the use of the *FILTER* class.

```
struct FILTER_SECTION{
    int numCoeff;
    double gainFactor;
    double *lastOutput, *lastInput, *a, *b;
    FILTER_SECTION(){
        a=b=lastOutput=lastInput=NULL;
    }
};
class FILTER{
private:
    int numSections;
    FILTER_SECTION *section;
};
```

loaded *FILTER* constructors to calculate the coefficients by other methods (impulse and step invariance, or matched-*z*). By truncating or rounding the coefficients to a given number of bits, the simulation engine would model the actual quantized response of an IIR structure.

Beyond this, no further modifications should be required. Simply call either *TransferFunc()* or *TimeStep()* as you would for analog filters.

PUT IT TO USE

This C++ class is easily adaptable to existing simulation software. You can also expand the source code to include arbitrary IIR design and simulation.

My application uses it to simulate several popular analog-filter types. You can use it to evaluate filters with sine- or square-wave inputs, which is helpful for determining whether a given filter meets the rise or fall time or attenuation specs for a particular application. ☒

Glenn Parker works for Eagleware and has been writing circuit design and simulation software for five years. He also teaches filter-design courses at several RF and microwave trade shows annually. You may reach him at glenn@eagleware.com.

SOFTWARE

Complete source code is available via the Circuit Cellar web site. For a more theoretical description of the techniques and equations used to develop the source code, a complete Mathcad worksheet is available from the author.

REFERENCES

- Oppenheim, V. and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
- Orchard, H.J. and A.N. Wilson, Jr., "Elliptic Functions for Filter Design," *IEEE Transactions on Circuits and Systems*, **44**, April, 1997.

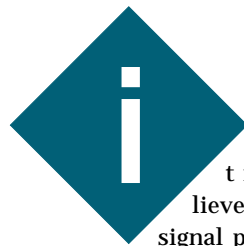
Breaking Nyquist

FEATURE ARTICLE

Gerard Fonte

Post-Sampling Antialiasing

You'd think someone would have figured it out by now: you can't get away with challenging an engineer. Gerard is set to disprove the common belief that DSP can't identify aliased signals. According to him, you can get around Nyquist!



It is generally believed that digital signal processing (DSP) cannot remove or identify aliased signals. For years, we've heard "Once the aliased signal gets digitized, it becomes indistinguishable from real signals and cannot be removed."

But, this statement isn't exactly true. There is a method for identifying and removing aliases after sampling, and it's even possible to identify and measure signals above the sampling rate.

THE PROBLEM

Aliasing is possible with any sampled system, including ADCs, switched-capacitor filters, sample-holds, and any other circuit that measures a signal at intervals. Aliasing doesn't occur with continuous-time circuits like analog filters, amplifiers, phase-locked-loops, frequency-to-voltage converters, or analog multipliers.

An alias is created by mixing (or beating) the sampling frequency and the sampled signal, hence another name for alias is beat frequency.

Suppose you want to measure a 3-kHz sine wave, and you sample it at 10 kHz. The beat frequencies are the sum and

difference of the signal and the sample rate—in this case, 13 kHz and 7 kHz.

You can see that lowering the sampling rate to 8 kHz gives beat frequencies of 5 kHz and 11 kHz. At a sampling rate of 6 kHz, the beat frequencies are 3 kHz and 9 kHz. In the latter example, the lower beat frequency is equal to the signal frequency (3 kHz), which is known as the Nyquist limit.

The Nyquist limit is always one half of the sampling rate. This description isn't the traditional definition, but it lets us look at Nyquist from a different point of view.

As you'll see, there's an easier way to examine some important aspects of the Nyquist sampling theorem. And from there, you'll see that the Nyquist limit can be sidestepped.

ELIMINATING ALIASING

Now, let's go back to measuring a 3-kHz signal at 10 kHz (samples per second). Suppose there is also a 7-kHz signal present. What happens?

Well, the 7-kHz signal beats with the 10-kHz sample rate and generates 17- and 3-kHz signals. Obviously, the 3-kHz beat frequency is the same as the real 3-kHz signal, but it's an alias. Clearly there's a problem: we can't tell the difference between a real 3-kHz signal and a 7-kHz signal.

The traditional method is to low-pass filter the signal to eliminate any signals (or signal components) above the Nyquist frequency. In this case, since the sampling rate is 10 kHz, the Nyquist limit (and filter cutoff) is 5 kHz.

However, filtering is more easily said than done. The 7-kHz problem signal is only about twice the signal of interest (3 kHz). A simple resistor-capacitor (RC) filter (one pole) reduces the 7-kHz signal by a little more than 6 dB (50%).

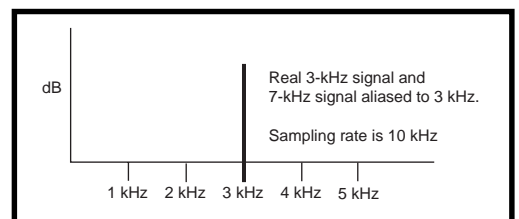


Figure 1—When sampling at 10 kHz, a real 3-kHz signal and a 7-kHz signal get mapped to the same spectral line of 3 kHz. The 7-kHz signal is aliased to 3 kHz. Traditional techniques are unable to separate the two different signals from each other.

In a digital system, a 6-dB reduction is equivalent to shifting the alias amplitude value one bit to the right. Most likely, this reduction won't help.

FILTERING VS. BANDWIDTH

The real world uses sharp-cutoff, low-pass filters of eight or more poles, which increases complexity. Additionally, the sampling rate is generally increased 4–10 times greater than the highest frequency of interest, which decreases bandwidth (relative to the theoretically possible).

Finally, the filters generally must be analog types (or digital filters with clock rates 10–100 times greater than the sample rate and with their own analog antialiasing filters).

Therefore, the performance of any conventional sampled system depends on the filtering of the signal before the signal is sampled. The closer to the theoretical maximum bandwidth you want (Nyquist limit), the closer to a perfect filter you need.

Realistically, if the highest signal of interest is 3 kHz, you may need a minimum sampling rate of 15–30 kHz. So, you'll need a faster (read: more expensive) ADC than what you might have initially expected.

THE SOLUTION

But, we can improve things. Let's reconsider the 3-kHz signal with 7-kHz signal, sampled at 10 kHz (see Figure 1).

As I stated, the 7-kHz signal generates an alias at 3 kHz and 17 kHz. Let's assume the DSP system has an internal Fast Fourier Transform (FFT) bandwidth from 0 to 5 kHz. So, you can ignore the 17-kHz signal.

But, you still have two signals (3 and 7 kHz) mapped to one frequency (3 kHz). The solution is simple: sample again at a different rate.

Let's try 9 kHz. The real 3-kHz signal is still mapped to 3 kHz since it's below the Nyquist limit. But, the 7-kHz signal is shifted from 3 to 2 kHz (see Figure 2). Comparing the FFT spectra of two signals at two different sampling rates enables us to identify aliased signals.

That's the fundamental idea behind post-sampling antialiasing.

Aliased signals track with the sampling rate, while real signals (below the Nyquist limit) remain fixed. In this way, aliased signals can be separated from real signals after sampling.

GOING FURTHER

Since there is a specific relationship between the aliased signal and the sampling rate, the aliased signal can be calculated. Let's look at an example from the DSP point of view.

What is the real frequency of a signal that appears at 3 kHz when sampled at 10 kHz, but shifts to 2 kHz when sampled at 9 kHz? Finding the answer is simple.

The 3-kHz beat frequency at 10 kHz means a real frequency of 13 or 7 kHz. The 2-kHz beat frequency at 9 kHz means a real frequency of 11 or 7 kHz. Since 7 kHz is common to both sampling rates, it must be the answer (i.e., the unknown alias frequency).

It's important to note that this simple procedure has measured and identified a signal above the Nyquist limit. But, you ask, suppose the additional frequency was 13 kHz instead of 7 kHz? What then?

Let's work it out. First, I cheated a little for simplicity. In the previous example, there were many more possible alias frequencies than I implied.

A 3-kHz alias signal from a 10-kHz sampled system could be 7, 13, 17, 23, 27, or 33 kHz, and so forth. There is actually an infinite number of possible signals that generate a 3-kHz alias.

The calculation is:

$$F_a = (n \times F_k) + F_s$$

or

$$F_a = (n \times F_k) - F_s$$

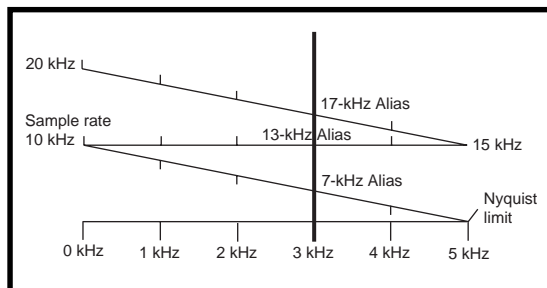


Figure 3—The Z map technique quickly illustrates where aliases will be mapped. You can see that the alias may increase or decrease with increasing frequency. Sometimes this characteristic causes confusion.

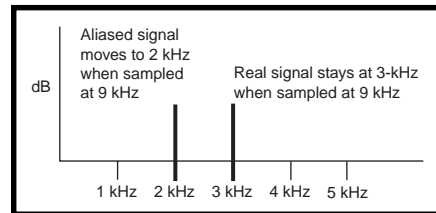


Figure 2—By sampling a second time at a different rate, the alias signal is separated from the real signal. The amount of alias shift is directly related to the change in rate.

where F_a is the alias frequency, F_k equals the sampling rate, F_s represents the signal frequency, and n is any positive integer.

There are exactly two aliases for each n , and as n increases, the alias changes by an identical factor. Also, note that the alias is only created when the signal frequency is greater than half the sampling rate.

Back to the problem of 3- and 13-kHz signals sampled at 10 kHz and then again at 9 kHz. Let's assume there is a perfect low-pass filter at 50 kHz to limit the number of possible aliases.

The 3-kHz signal creates no alias because it's less than half the sampling rate. It stays at 3 kHz when sampled at either 9 or 10 kHz. The 13-kHz signal creates an alias at 3 kHz when sampled at 10 kHz and an alias at 4 kHz when sampled at 9 kHz.

If you didn't know the additional signal was 13 kHz, how would you determine it? The procedure is the same, except that there are more than two pairs of possible alias sources.

At 10-kHz sampling, the 3-kHz alias could come from a signal at 7, 13, 17, 23, 27, 33, 37, 43, or 47 kHz. At 9-kHz sampling, the 4-kHz alias could come from a signal at 13, 14, 22, 23, 31, 32, 40, 41, 49, or 50 kHz.

The only common value between the two lists is 13 kHz. Therefore, the alias signal that creates a 3-kHz signal when sampled at 10 kHz, and a 4-kHz signal when sampled at 9 kHz, must be 13 kHz. This procedure has identified and measured a signal above the sampling rate.

Note that other frequencies create a similar pattern. In this particular case, 67 kHz (like 13 kHz) produces a 3-kHz alias

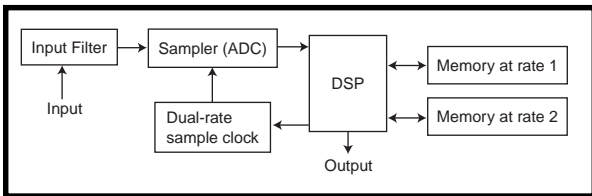


Figure 4—Here's a typical implementation of a system with a repetitive signal. There is no significant hardware difference from conventional systems, except that the sample clock must work at two speeds.

when sampled at 10 kHz and a 4-kHz signal when sampled at 9 kHz.

This situation is expected. If we don't band-limit the system, we're mapping an infinite number of frequencies onto the finite bandwidth of our system.

But, you can eliminate this problem, too. Sampling the input signal again at a third sample rate will resolve the 13-kHz and 67-kHz signals.

You just map out the possible aliases created at the multiple sampling rates and determine common values. In theory, I could resample as many times as needed to resolve, identify, and measure frequencies at any frequency desired.

Adaptive sampling techniques can be used as well. The second sampling rate (or third rate, etc.) can be chosen by the system in response to the spectrum analyzed.

The spectrum taken at one sample rate can be examined, and the

most appropriate second sample rate (or third, etc.) can be implemented to separate alias signals. Obviously, you'll need more smarts in the software.

LIMITATIONS

The more alias frequencies that are mapped onto real frequencies, the more complex the spectral comparison software needs to be. Suppose you had two real signals at 2 and 3 kHz: a 7-kHz signal that was aliased to 3 kHz when sampled at 10 kHz and to 2 kHz when sampled at 9 kHz.

In this case, the alias signal maps onto two real signals, which is different from the ideal cases I just presented. Since the alias can only add to a spectral line, the spectral-comparison software must use the lowest value as the best estimate for the real signal.

With lots of signals above the Nyquist limit, many aliases map onto real signals and the spectral comparison software becomes more complex. Trying to map DC-to-50 kHz onto a DC-to-5-kHz FFT bandwidth gives a ratio of nine alias signals for every real signal. It may be theoretically possible, but it's probably an exercise in futility.

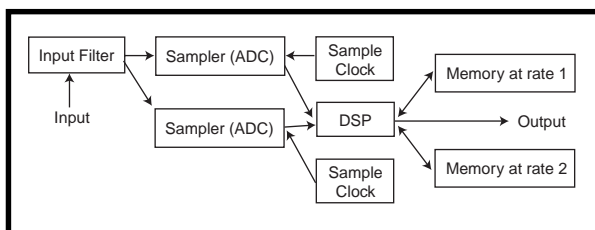


Figure 5—For nonrepetitive or one-shot systems, two clocks and two input samplers are needed. Note that the samplers could be sample-holds feeding a single ADC. The approach here is suitable for any application.

Up to now, I've examined discrete spectral lines and signals. However, noise is all spectral lines over some bandwidth. If the noise is limited to a bandwidth less than the difference of the sampling rates, it can be isolated.

Suppose there is a noise band from 7 to 8 kHz and a real signal at 2.5 kHz. We sample at 9 kHz and 10 kHz. Sampling at 10 kHz maps the noise band to 2–3 kHz, which overlays the 2.5-kHz signal. Sampling at 9 kHz maps the noise to 1–2 kHz, exposing the signal of interest. (If the 2.5-kHz signal is large when compared to the noise, it will still be measurable, but degraded, even when the noise is mapped onto it.)

Broad-band noise (5–20 kHz) does not appear to shift because another spectral line replaces every one that moves. Worse still, three times the noise is mapped onto the real spectrum (in this case) because there is a three-

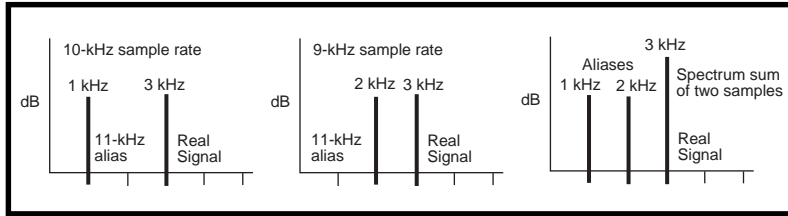


Figure 6—Summing the spectra (instead of just examining them) has the effect of dispersing the alias. This dispersion reduces the alias amplitude relative to the real signal.

to-one (alias to real) ratio when sampled at 10 kHz. I discuss noise more below.

ALIAS MAPPING

To help identify alias mapping, I use a method called Z mapping (which has nothing to do with the Z transform). The Z refers to what the map looks like (see Figure 3).

The base of the Z identifies the real signals. It goes from DC to the Nyquist limit (half the sampling rate). The return line goes from the Nyquist limit to the sampling rate.

The third line is two times the base values, the fourth line is two times the second line, and so on. Drawing a vertical line upward at any real signal

(base line) identifies the aliases mapped to that real signal.

IMPLEMENTATION

There are two basic methods of implementation. The first, shown in Figure 4, is like a conven-

tional A/D system with the addition of a variable sample clock.

The DSP software has to change slightly, with an additional FFT at the new sample rate and an FFT spectrum comparison routine. Since the sample rate affects the characteristics of the FFT, the spectrum comparison routine won't be completely trivial.

This method is suitable for systems with constant or repetitive signals (rather than one-shot measurements). For example, an oscilloscope implementation can sample a repetitive signal with even-numbered sweeps at one rate and odd-numbered sweeps at another.

The second method, shown in Figure 5, is to use to separate front-end

samplers, such as two sample-holds or two ADCs. The input signal is simultaneously sampled at two rates, which enables nonrepetitive one-shot signals to be measured. This method requires the same software changes as above.

REAL-WORLD CONSIDERATIONS

Let's look at the expected performance of a conventional system and the proposed system with real-world components, measuring from DC to 5 kHz with 12-bit resolution. To maintain this resolution, all aliases must be less than one bit at any frequency.

The conventional system requires a low-pass filter of 12 poles set at 5 kHz to reduce the alias at 10 kHz to less than 1 bit. In other words, there must be 72 dB per octave of low-pass input filtering, starting at 5 kHz.

Since there is measurable alias energy up to 10 kHz, the sampling rate must be twice that, or 20 kHz. In short, even with a very sharp input low-pass filter (which isn't trivial to implement), the bandwidth is still only 50% of the theoretical maximum.

The proposed system, however, has a much more flexible design. If you limit the input signal to 20 kHz, you need to low-pass 72 dB over two octaves (5–10 kHz and 10–20 kHz), not one.

You need a six-pole low-pass input filter set at 5 kHz to reduce the input aliases to less than one bit at 20 kHz. That setup limits the spectrum mapping to a ratio of three aliases to one real signal, and it makes the spectrum comparison routines fairly simple. (There's a tradeoff here: making the comparison routines more complex versus simplifying the input filter.)

Since you want the lowest alias-to-real ratio (stated as three to one), you need to have one sampling rate at the Nyquist limit of 10 kHz. The second sampling rate is arbitrary but should

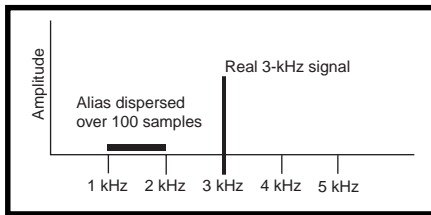


Figure 7—By summing the spectra of 100 samples, the aliases are dispersed by a like factor. The use of a swept clock looks promising.

generally be slightly less than the first sampling rate.

Let's choose 9 kHz. The sum of the A/D rates is 19 kHz, which is close to the conventional example of 20 kHz.

This example shows how input filters change with similar A/D rates. In this case, the input filtering changes from a 12- to a 6-pole requirement. That's significant because a 12-pole filter is more than two times harder and expensive to implement than a 6-pole filter. (Consider the differences between 6- and 12-bit ADCs.)

Or, you could stay with a 12-pole filter. The input signal is limited to 10 kHz (as specified above), and you can sample at 5 and 4.5 kHz, maintaining the spectrum-mapping ratio of three to one.

However, the aliases from 2.5 to 5 kHz are not discarded. Instead, they are identified and remapped to their proper spectral positions.

This approach means more software, but it's not difficult, and it provides 100% of the bandwidth of the theoretical maximum. So, you get twice the bandwidth of the conventional system.

MORE SAMPLES

Up to this point, I've shown you FFT spectra and removed or remapped spectral lines. But, suppose you summed the spectral lines (see Figure 6) so the real-signal amplitude increases while the alias amplitudes remain the same. Another way of saying this is that the energy of the alias is dispersed over two spectral lines.

The result leads to an interesting thought.

Suppose you sampled at many rates—say, 100. You'd see the alias energy automatically dis-

persed over 100 frequencies, which is a 40-dB reduction of the alias compared to the real signal (see Figure 7).

The dispersion of the alias energy relates directly to the number of different sample rates. Note that direct FFT spectral comparisons aren't needed. This approach appears suitable for ordinary digital filters.

Another interesting aspect of this method is that wide-band noise appears to be reduced (relative to the real signal), because when noise is summed, it increases by the square root of the number of summations.

The real signal increases directly to the number of summations. With 100 sample rates, noise can be reduced by 20 dB (compared to 40 dB for a discrete alias signal).

With this in mind, it becomes clear that using a swept-sample clock may be useful. Using this clock has the effect of sampling at many different rates.

It seems reasonable that, since the swept-sample clock is well defined and repeatable, you can make corrections to the computations (to compensate for the swept-sample clock). This way, you can reduce post-sampling aliasing and maintain low sampling rates, simple filtering systems, and simple sampling systems (see Figure 8).

Unfortunately, the math corrections aren't trivial. Work is slowly progressing along these lines, but the use of swept-sampling may hold significant improvements in usable bandwidth, alias reduction, and noise reduction.

GETTING AROUND NYQUIST

You can sidestep the Nyquist limit by multiple sampling of the input signal at different sample rates. Frequencies above the Nyquist limit, and even above the sampling rate, can be identified and measured, reducing input filter requirements and increasing effective bandwidth. ☐

The procedure of multiple input sampling for the identification and reduction of aliases has been patented by The PAK Engineers.

Gerard Fonte founded The PAK Engineers in 1991 and still works as Principal Engineer. You may reach him at gfonte@wzrd.com.

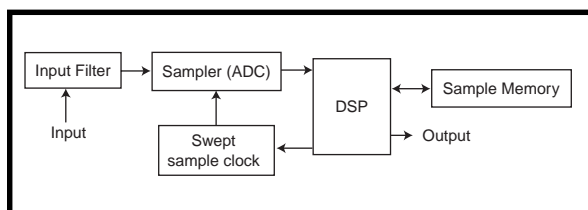


Figure 8—The hardware design for a swept-clock implementation is simple. However, the software must be significantly and fundamentally changed (and this isn't a trivial issue).

Digital Frequency Synthesis

FEATURE ARTICLE

Tom Napier

Some of the best inventions happen by accident. At least, that's how it is with Tom's project. While merely intending to beef up his NCO generator, Tom found a way to embed a low-cost, accurate tunable sine-wave generator, using just a PIC.



Sometimes topics for articles crop up by happenstance. I was designing a minimum shift keyed (MSK) transmitter to drive a 25-kHz ultrasonic transducer, and for simplicity, I wanted to use a small PIC chip as the transmitting modem.

I figured out several ways to generate the two output frequencies but one method seemed simpler and more general than the others. The frequencies were set by two constants and there seemed to be little limit to what values could be used.

"Ah!" I said to myself, "This design would work for any frequency shift keying system, not just MSK." That's when I realized that I'd just reinvented the numerically controlled oscillator (NCO).

Someone who read my two-part article about building an NCO generator ("Making Waves with an NCO," *INK* 89 and 90) asked if the PIC could emulate an NCO.

Well, the short answer is yes, providing you don't want to generate too high a frequency. I didn't give the matter any thought until I discovered that I had just created a design that was happily generating 25 kHz using a 6.144-MHz crystal.

I thought, if I give the PIC a higher crystal frequency—say, 20 MHz—and

can get it to output an 8-bit sample every 20 instructions, it will behave like an NCO with a 250-kHz clock crystal. That should be good for an output frequency of getting on for 100 kHz, which is better than most audio generators can do.

I followed up this idea and ended up with not so much a construction project, but more a method of embedding a low-cost but accurate, tunable sine-wave generator into almost any product. This design doesn't need a fancy NCO chip or a high-speed DAC. It uses an 18-pin PIC16C54 chip to drive a cheap, low-speed 8-bit DAC.

In a pinch, you could wire up a bunch of 1% resistors as an R-2R ladder and do without the DAC. The only other things you need are a low-pass filter and an output amplifier.

WHY IT WORKS

Let's recap some theory. The idea behind the NCO is this: Select a number proportional to your desired output frequency, and then add this number to an accumulator at regular intervals.

The number in the accumulator represents the phase of the output cycle. If you take its more significant bits, do a sine conversion and feed them to a DAC, you get one sine wave each time the accumulator wraps around.

The output frequency is a linear function of the number you are adding. The frequency precision is as good as that of the crystal driving the addition, while the frequency resolution can be made as fine as you wish by using a long enough accumulator.

The output from the DAC consists of steps that occur at a fixed rate. If you select a small frequency control number, you get a low-frequency output made up of many small steps and it will look quite smooth. As you select larger numbers, the output frequency rises, the number of steps per output cycle falls, and the output begins to look jagged.

Appearances are misleading. The sine wave is still there, but it's being distorted by the higher frequency aliases arising from the sampling process. With a good enough output filter, you get a clean sine wave at the specified frequency.

Listing 1—By using a 65-entry sine table and a “fixed” return address, a 16-bit NCO can be emulated with a 19-instruction-cycle loop.

```

; 19-cycle loop, 16-bit accumulator
      GOTO   DONE           ;start output
RETX: CALL   RETY
      GOTO   LOOP
RETY: RETLW  0             ;dummy to set return address
DONE: CALL   RETX         ;set up fixed return address
; look-up return reenters at this point
LOOP: BTFSS  PHASH,INV     ;test if inversion needed
      SUBWF  ZERO,0       ;invert output
      MOVWF  PORTB        ;output sample
      BTFSC  PORTA,DAT    ;test for change flag
      GOTO   NEW          ;get new frequency
LPI1: MOVF   FREQL,0       ;get current frequency
      ADDWF  PHASL,1      ;increment phase
      BTFSC  3,0          ;skip if no carry
      INCF   PHASH        ;propagate carry
      MOVF   FREQH,0      ;get frequency
      ADDWF  PHASH,1      ;increment phase
      MOVF   PHASH,0
      ANDLW  63           ;get sine table index
      BTFSC  PHASH,REV    ;test if reversal needed
      SUBWF  MAX,0        ;reverse index
      ADDWF  PC,1         ;jump to table
; one quadrant look-up table, 65 entries
      RETLW  128
      RETLW  131
      RETLW  134
; table continuation omitted

```

Nyquist’s Theorem says that, provided you don’t use fewer than two samples per cycle, a low-pass filter can take out the unwanted frequencies and leave a pure sine wave. Well, Nyquist was an optimist.

Two cycles is the theoretical minimum, and this assumes that you can build a brick-wall filter at half the sampling frequency. Three samples per cycle is a more realistic number.

In any case, Nyquist’s samples were infinitely narrow impulses, while the output of the DAC is a step waveform. Therefore, the output amplitude rolls off as the output frequency rises (see *INK* 89, p. 74, Figure 4).

The output frequency is equal to the step frequency multiplied by the number you set and then divided by the number of counts it takes to roll-over the accumulator. For example, if you have a 16-bit accumulator, it takes 65,536 counts to roll it over.

If you add any number 65,536 times a second, each unit you add represents 1 Hz. To get a 10 kHz output, add 10,000 at each step. Each output cycle would have just over 6.5 steps in it.

CAN A PIC EMULATE AN NCO?

Real NCOs have accumulators with 24, 32, or 48 bits and make an output step every 15 ns or so. It takes a fast and expensive DAC to keep up, but you can get output frequencies up to about 30 MHz.

If a ’16C54 really hustles, it can handle a 16-bit sum and a sine lookup

in 19 instruction times. With a 20-MHz crystal, the step rate is limited to 263 kHz. Pretty shabby by NCO standards, but it means you can generate accurate frequencies up to about 90 kHz.

There are two ways to approach the frequency-setting problem. One is to choose the accumulator length and the update frequency such that the proportionality factor between the wanted frequency and the frequency setting number is a small integer. That way, all the frequencies you set come out exactly right.

In my MSK modem design, this factor was 300 Hz per unit. For lab use, 1 Hz per unit would be handy, since reading and processing the user’s input would only require a BCD-to-binary conversion.

Or, you can use such a long accumulator that any frequency can be set with reasonable accuracy regardless of the update rate. That buys you some design flexibility at the price of more input processing and an output frequency which is rarely, if ever, exactly right.

In both cases the proportionality factor, F , is equal to the update rate divided by the length of the accumulator. If the PIC is driven by a crystal of C Hz and the loop requires L instructions, the effective update rate is:

$$F = \frac{C}{4AL}$$

The accumulator length, A , is a power of two, such as 65,536.

A PIC Trick

The usual way to make a look-up table in a PIC is to program a string of RETLW N instructions and then make a CALL to a subroutine that adds the index to the program counter. If you want a 19-instruction-time loop, you have to use a trick: Preload the two return address registers in the PIC with the address of the start of the loop. And at the end of the loop, jump into the table. Each table entry loads a constant into the W register and does a return. This puts you back at the beginning of the loop with a sine sample ready to output.

This process saves executing a call to get to the table and a jump to reenter the loop. It works because every time you execute a return, the first return address register is used and the second return address register is copied into the first. You get a free branch address without anything changing. The return address registers are loaded once at the beginning of the program by executing a call just above the loop code and then executing a further call and a return. This process leaves a fixed return address pending.

Listing 2—A 24-bit NCO can be emulated in 26 cycles. The 31-cycle loop shown here has a simple conversion factor and room for additions.

```

; three-byte accumulator, 26 cycles minimum
;start output
GOTO DONE
RETX: CALL RETY
      GOTO LOOP
RETY: RETLW 0 ;dummy to set return address
NOCRY: GOTO LP2 ;waste time if no carry
DONE: CALL RETX ;set up fixed return address
LOOP: NOP ;five spare instructions
      NOP
      NOP
      NOP
      BTFSC PHASH,INV ;test if inversion needed
      SUBWF ZERO,0 ;invert output
      MOVWF PORTB ;output sample
      BTFSC PORTA,DAT ;test for change flag
      GOTO NEW ;get new frequency

LP1: MOVF FREQL,0 ;get current frequency
     ADDWF PHASL,1 ;increment phase
     BTFSS 3,0
     GOTO NOCRY
     INCF PHASM,1 ;propagate carry
     BTFSC 3,2 ;skip if not zero
     INCF PHASH,1 ;further carry propagation
LP2: MOVF FREQM,0 ;get frequency
     ADDWF PHASM,1 ;increment phase
     SKNC
     INCF PHASH,1 ;propagate carry
     MOVF FREQH,0 ;get frequency
     ADDWF PHASH,1 ;increment phase
     MOVF PHASH,1
     ANDLW 63 ;get sine table index
     BTFSC PHASH,REV ;test if reversal needed
     SUBWF MAX,0 ;reverse index
     ADDWF PC,1 ;jump to table
; one quadrant look-up table, 65 entries
RETLW 128
RETLW 131
RETLW 134
; table continuation omitted

```

HOW HIGH CAN WE GO?

The maximum output frequency is limited by two things. One is the sheer processing speed of the chip. The highest frequency can't be higher than about a third of the number of times the loop is executed in 1 s. This fact limits a '16C54 to about a 90-kHz output.

The other limit is a function of the accumulator length and the resolution you want. The largest frequency control number that a 16-bit accumulator can support is about a third of 65,536.

Thus, if you want 1-Hz resolution, you can't go higher than a 22-kHz output. With 2-Hz resolution, you could reach 44 kHz and so on.

In a fit of bravado, I tested a circuit that could generate up to 100 kHz in

steps of 4 Hz. It worked, but it needs a pretty good filter to pass 100 kHz but stop the alias at 163 kHz.

I doubt if the program loop for a 16-bit accumulator can be shorter than 19 instruction cycles. Even that takes PIC trickery (see sidebar, "A PIC Trick"). The loop will be longer if you want phase or frequency modulation.

The crystal frequency can't be higher than 20 MHz. Since $C = 4ALF$ and A is a largish power of two, the crystal frequency has to be divisible by a power of two if the control factor is to be an integer. Ideally, it would be a stock value, but in a production environment, any oddball crystal frequency will work.

The other approach uses a 24-bit or higher accumulator. This technique

obviously requires more instructions per loop, because in a three-byte addition you have to allow for the once-in-a-blue-moon occasion when adding the low bytes propagates a carry into both the middle byte and the high byte.

The update rate is going to be at most about 190 kHz, so you won't be able to generate much over 60 kHz. On the other hand, the resolution will be better than 0.01 Hz, so any frequency you please can be set to that accuracy.

I stumbled on a good compromise: use a 20-MHz clock crystal and make the loop 31 instruction times long. If you multiply the wanted frequency by 104, you get a number that sets the output to within 180 ppm, which may be better than the error in the crystal frequency. Since the basic three-byte addition and table look-up takes 26 instruction times, you have five spare instructions to modulate the output if you want to.

You may be able to use a much shorter accumulator. If, as I did in my

MSK modem, you want a limited number of frequencies that are closely related, you can use an 8-bit accumulator. This setup allows any frequency of the form:

$$\frac{N}{256} \times \text{loop repetition rate}$$

On the other hand, if you want to set a low frequency with very high resolution, you can make the accumulator 32 or 48 bits long.

THE NCO CODE

Apart from initialization and the user input routine, the code consists of the sample loop and a sine look-up table. The table contains 65 entries and specifies one quadrant of a sine wave. If the two highest bits of the phase are both zero, the table is used directly. If the most significant bit is a one, the table output is inverted.

When the next to most significant phase bit is a one, the table index is

Listing 3—This practical 31-cycle loop shows one way of applying biphase modulation to the output. The full listing (NCOEMU.ASM) is available via the Circuit Cellar web site.

```

; 24-bit accumulator, biphase modulation via bit 3 of port A
LOOP: NOP                ;two spare instructions
      NOP
      BTFSC  TEMP,INV    ;test if inversion needed
      SUBWF  ZERO,0     ;invert output
      MOVWF  PORTB      ;output sample
      BTFSC  PORTA,DAT  ;test for change flag
      GOTO   NEW        ;get new frequency
LP1:  ADDWF  PHASL,1    ;increment phase
      BTFSS  3,0
      GOTO   NOCRY
      INCF  PHASM,1     ;propagate carry
      BTFSC  3,2       ;skip if not zero
      INCF  PHASH,1    ;further carry propagation
LP2:  MOVF   FREQM,0    ;get frequency
      ADDWF  PHASM,1    ;increment phase
      BTFSC  3,0
      INCF  PHASH,1    ;propagate carry
      MOVF   FREQH,0   ;get frequency
      ADDWF  PHASH,1   ;increment phase
; biphase modulation input
      SWAPF  PORTA,0   ;put modulation in bit 7
      XORWF  PHASH,0   ;mask inversion bit
      MOVWF  TEMP     ;save inversion bit
      MOVF   PHASH,1
      ANDLW  63        ;get sine table index
      BTFSC  PHASH,REV ;test if reversal needed
      SUBWF  MAX,0     ;reverse index
      ADDWF  PC,1     ;jump to table
; one quadrant look-up table, 65 entries
      RETLW  128
      RETLW  131
      RETLW  134
; table continuation omitted

```

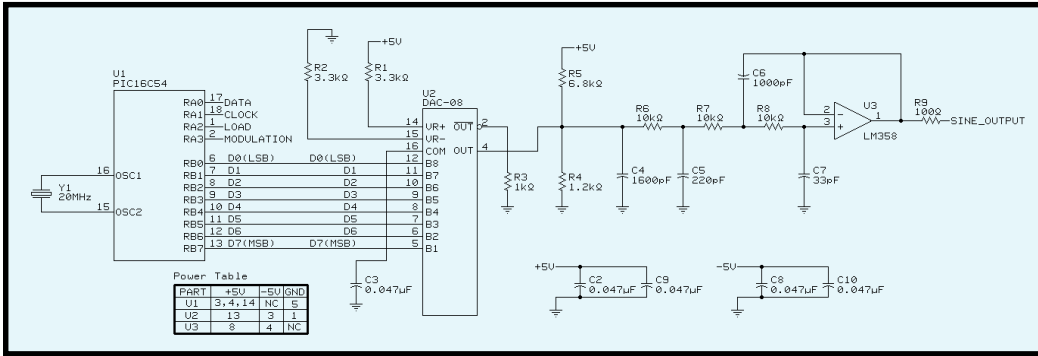


Figure 1—A 16C54, DAC, filter, and output buffer combine to make a tunable audio-frequency generator that you can drop into any project.

THE NCO HARDWARE

The hardware consists of the PIC, an 8-bit DAC, a low-pass filter, an output amplifier, and some method of loading the

desired frequency into the PIC. Figure 1 shows a typical system.

Some applications won't need all of it. My ultrasonic modem used a port bit to select one of two preset frequencies. It didn't need the low-pass filter since the frequencies being generated were both within the pass-band of the transducer and the alias frequencies were well outside it.

The hardware consists of the PIC, an 8-bit DAC, a low-pass filter, an output amplifier, and some method of loading the

The DAC input is in offset binary—that is, 1 is negative full scale, 128 is zero, and 255 is positive full scale. The table entries are seven-bit numbers from 128 to 255, so the result of subtracting them from a register containing zero is the negation of the table entry. Two PIC registers are preloaded with constants since the 16C54 has no immediate subtraction instruction.

My 19-instruction-time loop using a 16-bit accumulator appears in Listing 1. It tests a bit of Port A, enabling the user to tell the PIC that it's time to load a new frequency. Reading a five-digit BCD frequency and converting it into binary takes only microseconds.

The output voltage stays fixed during this, but in many cases the effect isn't noticeable. Reading a five-digit BCD frequency and converting it into binary takes only microseconds.

Listing 2 shows the more practical 31-instruction loop using a 24-bit accumulator. Listing 3 shows one method of incorporating biphase modulation via a bit of Port A. In all three listings, only the main loop is shown.

If you only want audio frequencies, the filter needs to be little more than a capacitor across the DAC's output resistors out to get a clean output. In the 50-kHz range, you need a filter that cuts off sharply above 50 kHz.

Figure 1 shows a four-pole quasi-Butterworth filter with a cutoff at 70 kHz. This filter approximates a Butterworth filter by combining a single-pole filter (the capacitor across the DAC output) with a three-pole Sallen and Key-style filter.

The two sections need to be isolated from each other, either by buffering the DAC output or, as shown, by making the resistors in the three-pole section about ten times larger than the DAC load resistor. The output amplifier doesn't need to be anything fancy because it only has to provide unity gain up to perhaps 500 kHz.

To set the frequency, I wired a five-digit thumbwheel switch to three 8-bit shift register chips as shown in Figure 2. The switches are ignored until the user presses the update button. Feedback from the output stage of the shift register debounces the button.

One bit from the PIC port keeps the shift register in Load mode. When the update button is pressed, the PIC puts the shift register into its Shift state. The bit that was sampling the

button then becomes the serial data input pin.

A third port pin supplies the clock to the shift registers to read all five switches. (The fourth pin of the 4-bit Port A provides a modulation input.)

Since the highest BCD digit takes values from 0 to 5, only 19 bits are read in. Three 19-byte look-up tables store the three-byte weight of each bit.

When a one bit appears at the input, the corresponding three-byte weight is looked up and added to the frequency register. If you embed this "NCO" in a larger system, you may be able to arrange this computation to be done elsewhere.

If you build this configuration, it is worth setting the frequency to 40,330 Hz, which gives a

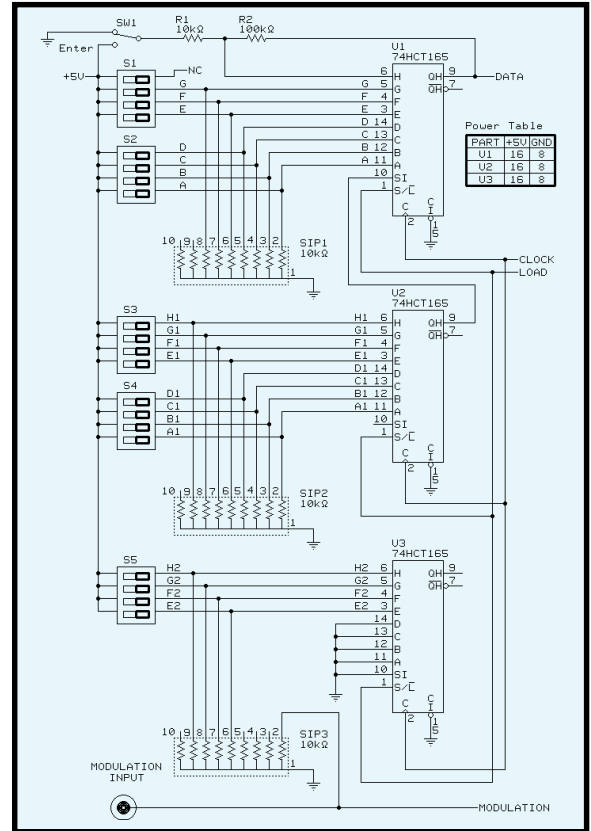


Figure 2—One way to tune the NCO is to connect five thumbwheel switches to some shift register chips.

frequency setting number that's close to a power of two so the DAC output is almost stationary. Compare the filtered output to the raw output on pin 2 of the DAC to see how good a job the filter is doing.

The DAC-08 is a former PMI part now made by Analog Devices. The DAC0800 from National Semiconductor is a direct replacement.

The DAC's outputs sink current to the negative supply. The currents vary in opposite directions as the input code changes. Both have full-scale values equal to the reference current (nominally 2 mA). If the reference current flowing into pin 14 changes, the output amplitude and its DC level change, too. This circuit's output is about 1.5 Vp-p.

The DAC-08 does need a negative supply—ideally, -15 V. If you can accept a limited output swing, -5 V will work.

ADDING MODULATION

The spare instructions in the 31-cycle loop let you add input tests to select either of two preset frequencies and generate FSK signals. The upper

two bits of the accumulator determine which quadrant of the waveform is used, so you can generate quadratic phase modulation by modifying these bits.


A 28-pin PIC would give you an extra 8-bit port. The number applied to it could be added to the phase register or to the frequency register to generate virtually continuous phase or frequency modulation.

The DAC output is proportional to its reference current. Modulating this current produces amplitude modulation of the output.

These techniques are a convenient way of generating low-frequency communication signals, and they can also be applied in the musical field.

LOWER FREQUENCY, LOWER COST

My earlier article described how a PIC, a specialized NCO chip, and a fast DAC could generate up to 10 MHz in a benchtop unit. But if you need lower frequencies, this article may give you food for thought. With a parts cost under \$10, a PIC and a DAC may be just the answer when you need a high-

precision adjustable-frequency sine wave below 50 kHz. 

Tom Napier has worked as a rocket scientist, health physicist, and engineering manager. He has spent the last nine years developing spacecraft communications equipment but is now a consultant and writer. In his free time, he develops neat test instruments, debunks pseudoscience, and writes in Forth on an Amiga 3000.

SOURCES

PIC16C84

Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 786-7277
www.microchip.com

DAC-08

Analog Devices
(617) 329-4700
Fax: (617) 329-1241

DAC0800

National Semiconductor
(408) 721-5000
Fax: (408) 739-9803

- 44** **Nouveau PC**
edited by Harv Weiner
- 48** **Networking with DeviceNet**
Part 2: A Weather-Station Application
Jim Brady
- 55** **Real-Time PC**
The Need for Speed
RTOS and PC/104
Ingo Cyliax
- 61** **Applied PCs**
RF and Micros
Part 2: A Low-Power System
Fred Eady



Photo courtesy of Derivation Co.

EMBEDDED

OCTOBER 1998

HIGH-SPEED A/D BOARD WITH DSP COPROCESSOR

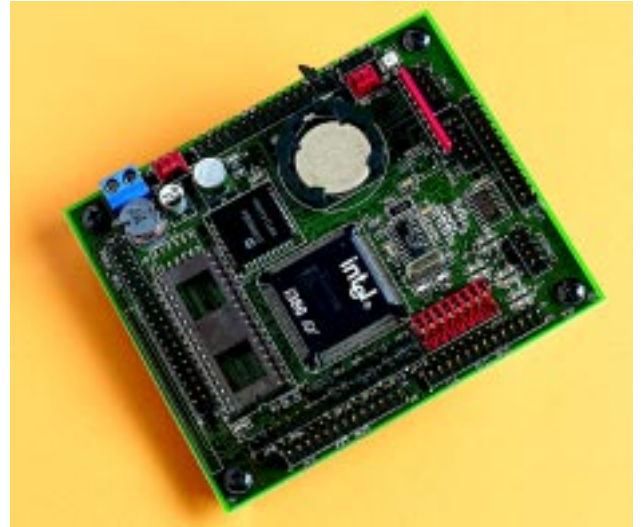
Datel Systems has announced the **PC-430K**, a high-speed ISA ADC-DSP coprocessor data-acquisition board. With two simultaneous-sampling 5-MHz ADCs, the PC-430K is ideally suited for continuous Fast Fourier Transform (FFT) processing, communications-receiver signal collection to disk, and simultaneous graphics display of spectral data.

Application areas include signal recovery from noisy channels, harmonic distortion analyzers, and vibration/resonance filter systems. The PC-430K can also be used for high-speed mapping and imaging, satellite channels, astrophysics, and seismology. Other uses include biomedical signals, array processing, control systems, simulators, engine analyzers, aerodynamics, and vehicle system applications.

The PC-430K acquires two analog input channels, and digitizes and stores them in local memory, while DSP math processing and data transfer are done concurrently. The system is designed for preprocessing seamless A/D datastreams to mass storage. Its timer/counter uses an onboard crystal oscillator or an external timebase for precision phase tracking. The ADC passes data directly to a FIFO memory that decouples the precision timing of the A/D section with the block transfers governed by the DSP. Some of the advanced features of the onboard DSP include single-cycle fetch and execution, parallel instructions, zero overhead of looping instructions, software variable wait states, block repeat, and 64-word internal instruction cache memory.

The PC-430K sells for **\$4945** in single quantities.

Datel Systems, Inc.
(508) 339-3000
Fax: (508) 339-6356
www.datel.com



C-PROGRAMMABLE CONTROLLER

The Z-World **ZB4100** is a board-level C-programmable controller that features an Intel '386EX processor running at 25 MHz. With 36 programmable DIO lines, one RS-232 channel, one RS-232 or RS-485 channel, and a parallel port, the ZB4100 provides all the interfaces needed for a wide variety of control applications. Its PLCBus support enables the user to expand these interfaces with their line of 4-bit expansion boards, which include digital I/O, relays, and D/A conversion. The board is also available with up to 512-KB SRAM and 512-KB flash memory, a remote programming port, battery-backed SRAM, and a real-time clock.

Programming is accomplished via Z-World's multitasking Dynamic Cx86. Based on C, Dynamic Cx86 offers keywords, software drivers, and library functions that meet the needs of control-system developers. Dynamic Cx86 integrates an editor, compiler, and interactive debugger in one package and eliminates the need for third-party debuggers or ICES.

The developer's kit for the ZB4100 includes a development board, aluminum baseplate, power supply, programming cable, and manual with schematics. Because the development board is programmed via a remote programming port, both serial ports are available for the application's use.

The ZB4100 is priced at **\$245** in quantities of 100.

Z-World
(888) 362-3387
(530) 757-3737
Fax: (530) 753-5141
www.zworld.com

DSP DEVELOPMENT SOFTWARE

IDE6000, a software development environment for the TMS320C6x DSP, is available under Windows 95 or NT. This GUI-based development environment enables users to configure hardware systems and manage software projects via wizards and dialog boxes. Configuration information provided to other elements of the IDE makes the development environment become system aware. This awareness extends to all board resources including processors, memory, host interface, and the I/O subsystem.

A choice of debugging environments is available with IDE6000, ranging from the Texas Instruments XDS510 with DB60, to Code Composer running native on the board. When the IDE6000 operates in native mode, the need for an external XDS debugging system is removed. All debugging systems are launched from within IDE6000 with only a few button clicks.

Project Make is a utility that lets you control software revisions and simplify code building by abstracting the user away from specific make file and linker issues. TI's C compiler is fully integrated into Project Make, and the user can select command-line compile options using a dialog box. An integral editor within Project Make enables the user to jump directly to any



errors found at compile time. Or, using the simple customization utilities, the developer can integrate his or her favorite editor.

Other building blocks include host interface libraries for host-to-DSP communication and utilities for configuring and accessing onboard resources (e.g., SCBus, MVIP, and flash memory). A download monitor is provided for interactively debugging the DSP and host system.

Two system test options are provided as well. Interactive Test, a GUI-based utility, enables the developer to select board features (e.g., memory or processor) and perform a test to confirm that they are functional. Built-In Test is used to test board functionality on startup. It is normally blown into flash memory and runs on startup. A file of test results is generated for the user to interrogate.

The IDE6000 is priced at **\$1000** per development seat with multiuser and site licenses also available.

Loughborough Sound Images, Inc.
(781) 860-9020
Fax: (781) 860-0083
www.lsi-dsp.com

COMPACT CPU MODULE

Intel's **PC104i** is a compact, low-profile, low-power CPU module that works with S-MOS Systems' '586 and '486 Card-PC, or Cell Computing's Pentium and '486 CardPC. Card-PCs have a standard pinout configuration and an EASI bus connector that make it easy to upgrade this PC/AT system by replacing one module with another (e.g., from a '486DX4 to a '586 module).

The PC104i has a single PC/104 stack with IDE hard disk, floppy disk, and simultaneous CRT and LCD video. Other features include VGA or CGA video, two serial ports, LPT parallel port, PS/2 keyboard and mouse, flash (disk or memory), watchdog timer, and a CMOS/clock battery backup. The module offers 5-V-only operation (3.3 V generated onboard), power management, 90° connectors option, optional 2-MB flash memory, and stackable 16-bit PC/104 bus expansion. PC104i is compatible with Windows 3.11, 95, and NT, as well as DOS and ROM-DOS.

Pricing starts at **\$250** in OEM quantities.

Intelc Technologies, Inc.
(847) 517-1000 • Fax: (847) 517-1001
www.intelec-tech.com



Nouveau **IPC**

DATA-ACQUISITION BOARD FOR CompactPCI

Analogic has introduced the **CPCI-14-1**, a high-speed, dual-channel, analog input board for CompactPCI.

Its superb spectral characteristics position it for high-performance applications like I/Q quadrature demodulations, communications, radar, medical MRI receivers, and event capture.

The CPCI-14-1 is designed to provide the highest possible signal-to-noise ratio and spurious-free dynamic range at sampling frequencies from 1 to 10 MHz in a 0–70°C range. Performance levels of 75-dB SNR and 90-dB SFDR are easily achieved with input signals as high as 5 MHz.

As an optional means of high-speed transfer, the CPCI-14-1 includes a DSP link port that provides a direct interface to a SHARC-based DSP board. The SHARC link port can be used for applications that require continuous uninterrupted data transfers and real-time DSP.

The 3U Eurocard format and its vertical installation in a protective card cage provide easy user access, excellent heat dissipation, and secure mounting—features that are required in rigorous industrial and mobile applications.

Prices start at **\$2500** each in small quantities.



Analogic Data Conversion Products
(800) 446-8936
Fax: (781) 245-1274
www.analogic.com

*Nouveau*IPC



AMD 5X86 SINGLE-BOARD COMPUTER

The **Little Monster VI** features a 32-bit, low-voltage AMD 5x86 CPU that runs up to 133 MHz on less than 7 W. Its 32-bit PCI and 16-bit ISA buses enable the computer to drive four PCI and eight ISA peripheral boards.

The Little Monster VI includes 16 or 32 MB of EDO DRAM (expandable to 96 MB), 256 KB of level-2 fast asynchronous cache memory, controllers for flat-panel VGA, PCI-enhanced EIDE, and floppy-disk support for 2.8-MB floppy drives. Also onboard are a standard keyboard port, four serial (16550) ports, two bidirectional EPP- and ECP-compatible parallel ports (IEEE-1284 complaint), and a watchdog timer.

Options include a CompactFlash socket onboard and an advanced resistive pen-and-touchscreen controller. Peripheral boards (PCMCIA, USB, sound, etc.) stack on top of

the Monster VI mezzanine-style to make the entire system as compact as possible. For enhanced reliability, the peripheral boards connect via leaf-spring connectors.

The Little Monster VI comes with SystemSoft BIOS to enable booting from a floppy disk, hard drive, CompactFlash, or PCMCIA device. The Monster supports Windows 95, NT, and CE in temperatures ranging from -25 to +70°C. The board size is 4" x 7".

The Little Monster VI starts at **\$400** in OEM quantities.

Zykronix, Inc.
(303) 799-4944
Fax: (303) 799-4978
www.zykronix.com

Nouveau PC

Networking with DeviceNet

Part 2: A Weather-Station Application

Think programming a DeviceNet interface in C++ is tough? Jim disagrees. With its excellent response times and adequate program size, Jim gets the same excitement writing code for a fast 32-bit CPU as he got from his '67 Camaro.

If you like programming as much as I do, you're in for a real treat with DeviceNet. The DeviceNet specification is fully object-oriented, with each object described in terms of attributes and services.

These items correspond to C++ class data and member functions, so if you use C++, all you have to do is understand the specification and translate it to code. Just make sure you have some strong coffee on hand when tackling the tricky parts. I had the most trouble—err...fun—with connection states and fragmented messaging.

Let's cover the PC/104 hardware first. After surveying the many processor boards available for PC/104, I went with the Micro/sys SBC1386, a 25-MHz '386EX board, shown in Photo 1. It comes with BIOS and a DOS run-time environment that runs the application out of RAM. That way, you don't need a special library and linker to generate ROMable code.

The board also includes the Borland remote debugger in flash memory. It's nice to be able to send the program to the

board at 115 kbps, set some breakpoints, and let 'er rip. My program is written entirely in C++ using the Borland compiler (large memory model).

A lot's been said about the poor suitability of C++ for real-time embedded development. But, it's more than adequate for a fast-response DeviceNet interface.

The program weighs in at 45 KB of code space, including the weather-station application code. This size is comparable to DeviceNet interfaces I've done using standard C with small CPUs. I'll show you some performance measurements later on.

CAN CHIPS

The next order of business is picking a CAN controller. Table 1 compares peripheral-type CAN controllers. I went with the Intel 82527 because I like having individual mailboxes for each message type rather than one big FIFO for all of them. It's more modular.

The Siemens parts also work this way. They have 15 or 16 mailboxes—plenty for

the DeviceNet predefined connection set, which has 10 connections.

A FIFO is good if you're concerned with the master beating your door down with high message rates. But at some point, your code will run out of steam anyway.

The 82527 has five operating modes. Only mode 3 (nonmultiplexed asynchronous) makes sense for a PC/104 interface. I'd prefer faster 16-bit transfers, but the 82527 in mode 3 is limited to 8 bits.

The PC/104 bus has the same timing as the ISA bus, and it takes a whopping 720 ns for an 8-bit read or write. This glacial pace is actually good because it doesn't exceed the rather long cycle and access times of the 82527.

In mode 3, at maximum clock rate, the 82527 has a 288-ns access time. If you use a fast bus, you need to accommodate this slow interface. Intel's web site has app notes for interfacing the 82527 to a various processors.

I use a PAL to generate the R/W select line and the chip selects. To make sure the

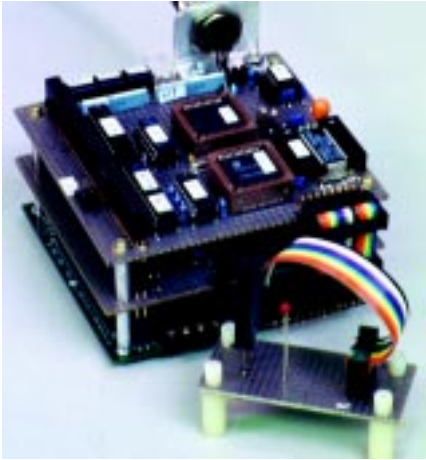


Photo 1—The PC/104 weather station is entirely powered from the DeviceNet bus. The weather station board is sandwiched between the '386EX CPU board on bottom and the DeviceNet interface on top. The humidity transducer and thermistor are on the small board in front.

R/W line remains stable at the end of a bus cycle, the line is latched by an RS latch in the PAL.

MEMR sets the latch and MEMW resets it. The PAL design source file is available via the Circuit Cellar web site. The only glue logic is a couple of inverters to delay the 82527 chip select to make sure it doesn't go low until after R/W is valid.

With no video board in my PC/104 stack, there's plenty of memory space for the CAN controller's 256 bytes. I went with A0000.

In mode 3, the 82527 provides one I/O port. To get enough I/O for all my switches and LEDs, I added an 82C55A at memory address A1000. That gave me plenty of I/O lines, including enough for a four-wire serial interface to the ADC on my weather-station board.

CHIP SETUP

The 82527 has 15 mailboxes for CAN messages, each with 15 registers. Setting up a mailbox requires telling it what its message identifier is and if it is send or receive. Done properly, your program only gets an interrupt for a message directed to your device.

The 82527 also has a group of registers that control message filtering, interrupt masking, data rate, and sample timing. There are some tricky ones that set the sample point within a bit time as well as the limit on how much that sample point can jump around.

There is a tradeoff—you want to let it jump as much as possible to accommodate oscillator tolerance, and you also want the sample point to be close to the end of the bit time to accommodate long cables. But you can't allow it to jump so much that it goes past the end of a bit time.

After a lot of calculation, I ended up sampling at 87% of the way through a bit time, with the jump limit (SJW) equal to 12% of a bit time. That accommodated the worst-case cable length, with a jump width still large enough to handle crystal errors of about $\pm 0.2\%$, which is plenty for any crystal. The *Intel 82527 Architectural Overview* provides information for this calculation.

REAL TIME

I can't help but be excited about writing code for a fast 32-bit CPU after designing 8-bit systems for years. The feeling of power is like the feeling I got from my first car, a '67 Camaro with a 327 engine.

To make sure the 18.2-Hz BIOS clock interrupt wouldn't hurt me, I measured its duration by looking at how big a chunk it took out of a tight loop that pulsed an I/O pin. According to my scope, it is just 56 μ s, including the time it takes to run my own timer interrupt at INT 1C, which is chained to the BIOS clock interrupt. I use this timer to update my DeviceNet connection timers.

When a DeviceNet message arrives, the 82527 pulls IRQ5 high. According to

Intel, the '386EX has a worst-case interrupt latency of 63 clock cycles, or $\sim 2.5 \mu$ s at 25 MHz, neglecting wait states.

So, my DeviceNet interrupt handler has to wait for a maximum of 58.5 μ s (i.e., 56 + 2.5) before it runs. This situation happens when a DeviceNet message comes in just after a BIOS clock tick.

The DeviceNet interrupt handler in Listing 1 reads the message-length byte to find out how long the message is and then reads only the data bytes it needs to. Most DeviceNet messages are well under 8 bytes. The most frequent message, the I/O Poll Request, has no data bytes at all!

By the way, check the disassembled machine instructions with your debugger to make sure functions like peekb() are getting expanded inline. Depending on compiler settings, they may not be. For an 8-byte message, the duration of my DeviceNet interrupt handler is 100 μ s with peekb() inline or 160 μ s otherwise.

These timing measurements show there's still plenty of time left for processing messages. DeviceNet recommends a response time of 1 ms for I/O Poll messages and 50 ms for Explicit messages. These measurements also show that a faster PC/104 bus wouldn't help much. Out of the 100- μ s total time for the interrupt handler, only about 6 μ s (eight bytes at 720 ns per bus cycle) is spent doing bus transfers.

	Intel 82527	Philips SJA1000	Siemens SAE 81C90	Siemens SAE 81C91
Package	PLCC 44 QFP 44	DIP 28 SO 28	PLCC 44	PLCC 28
Parallel CPU Interface	8-bit multiplexed 8-bit nonmultiplexed 16-bit multiplexed	8-bit multiplexed	8-bit multiplexed	8-bit multiplexed
Access time	288 ns	45 ns	120 ns	120 ns
Serial Interface	SPI, 8 MHz	None	4 wire, 5 MHz	4 wire, 5 MHz
I/O Ports Organization	1 or 2 eight-bit ports 15 mailboxes; 1 is double-buffered	None 64-byte FIFO	2 eight-bit ports 16 mailboxes	None 16 mailboxes
Identifier mask registers	1 global for mailboxes 1-14, and 1 special for mailbox 15	1 global	None	None
Identifier match registers	1 per mailbox	1 global	1 per mailbox	1 per mailbox
Message timestamp	No	No	Yes	Yes
Max. DC current	50 mA	15 mA	30 mA	30 mA
Approx. price	\$7.50	\$7.30	\$6	\$5.30

Table 1—Now you can compare various peripheral-type CAN controllers. The Philips device stores all messages in a FIFO, while other devices store messages in mailboxes based on their identifier.

If your CPU is slow, an easy way to get the 1-ms response time for I/O Poll messages is to put the data you want to send in the CAN chip's mailbox ahead of time, ready to go. When an I/O Poll request comes in, immediately tell the CAN controller to send it. With this strategy, I measured the weather station's I/O Poll response time at a worst case of 140 μ s.

I later changed the code to be consistent with my priority-based event handler, which runs in the main loop. My DeviceNet interrupt handler puts the message into a buffer, sets a bit in a 16-bit event-word to indicate a message is in, and exits.

The bit's position within the event-word determines its priority. When the main loop detects this bit and no higher priority bits exist, it calls the link consumer to consume the message, gets the data from the Assembly Object, and calls the link producer to produce the message. This orderly approach lengthens the I/O Poll response time to 340 μ s, which is still plenty good.

MESSAGE FLOW

Figure 1 shows message routes in the system. Explicit and I/O Poll messages come in through their respective mailboxes. Explicit messages are routed via the path specified in the message and can access almost any object in the device. I/O Poll messages grab preselected data from a buffer in the Assembly object and quickly send it.

The weather-station sends three bytes—device status, temperature, and humidity. I can send more data by adding it to the existing assembly or creating a second assembly. The device manufacturer determines which data goes into the assemblies.

At the top of Figure 1 is the unconnected port, which the master uses to allocate the connections it wants to use. Technically, connections don't exist prior to allocation.

This situation implies using C++ dynamic allocation. Although you can do this, I chose to create static objects at the beginning of `main()` and use the constructor to set the initial connection state to nonexistent.

CONNECTIONS

Connections have states other than nonexistent and established, and some are unique to one or the other connection. This setup is so confusing, I made a state transition diagram. Figure 2 combines the

Listing 1—The interrupt handler for DeviceNet messages copies the message from the 82527 into a buffer, saves the length, and frees the 82527 for the next message. The run time is 100 μ s.

```
#define CAN_BASE 0xA000
UCHAR global_CAN_buf[10];
UINT global_event;

// Handles receipt of incoming DeviceNet messages
// The three dots are required in C++ mode
void interrupt far can_isr(...)
{
    UCHAR i, int_source, addr, mailbox, length;

    int_source = peekb(CAN_BASE, 0x5F); // read interrupt source
    if ((int_source < 3) || (int_source > 7)) return;

    mailbox = int_source - 2;
    for (i=0; i < 10; i++) global_CAN_buf[i] = 0;

    // compute address of config register in mailbox of interest
    addr = 6 + (mailbox << 4); // multiply by 16
    length = peekb(CAN_BASE, addr); // read message length
    length = length >> 4;
    global_CAN_buf[9] = length; // save message length
    for (i=0; i < length; i++){ // move message from 82527
        addr++;
        global_CAN_buf[i] = peekb(CAN_BASE, addr);
    }
    addr = 1 + (mailbox << 4); // point to control 1 reg.
    pokeb(CAN_BASE, addr, 0x55); // clear INT_PENDING bit
    addr--; // point to control 0 reg.
    pokeb(CAN_BASE, addr, 0xFD); // clear NEWDAT
    global_event |= 0x0001 << mailbox; // set bit in global_event
    outp(0x20,0x20); // nonspecific EOI
}
```

behavior of both types of connections, using colors to tell them apart.

When the master allocates the Explicit connection, the connection simply transitions to the established state and it's ready to use. The connection timer starts at 10 s.

If it times out, the connection goes to one of two possible states depending on whether the connection is in autodelete or deferred-delete mode. In autodelete mode, if it times out, it's gone. In deferred-delete mode, it stays around and goes back to the established state if a message comes in.

The I/O connection, when allocated, goes to the configuring state. In this state, it cannot process I/O messages and must wait for the master to set its expected packet rate via the Explicit connection. Then it is in the established state and can begin handling I/O Poll requests.

TIMERS

For each connection, you need a time-out timer. You also need a timer for sending fragments. The BIOS clock is handy, but who wants to deal with 18.2 Hz?

With the Micro/sys board, the 25-MHz system clock is divided by 21 to drive timer 0 in the '386EX. Timer 0 further divides by 65,536, producing 18.2 Hz. Loading 0xE884 into the timer 0 count register resulted in BIOS clock interrupts at a more friendly rate of 20.0 Hz.

The connection time-out time depends on the expected packet rate, which is set by the master. When a DeviceNet message comes in, I reload the timer for that connection, and my timer interrupt handler then decrements it at a 20.0-Hz rate. If it reaches zero before another message comes in, the connection times out.

ANALOG INPUT POINT

The Analog Input Point in the DeviceNet object library models an analog sensor. Listing 2 shows some code for this class.

The specification defines eight attributes, many of which are optional. I implemented the ones for sensor value, sensor status, and data type. The data type tells the master whether the value is an integer, float, or what. For the weather

station, I use an unsigned char that corresponds to a data type of 2.

My Analog Input Point class implements the DeviceNet Get Attribute Single service using a member function. Thus, the master can read any of the three attributes using an Explicit message.

These attributes aren't settable, so my class doesn't have the Set Attribute Single service. In the future, I may allow the master to set the data type to a float, switching my sensor value to floating point.

IDENTITY OBJECT

Every device must be able to give its name, rank, and serial number. The Identity object holds this information. It also keeps track of device state and does device resets.

There are two types of resets. Type zero simulates an off/on power cycle. To do this, I send a response back to the master and suspend writes to my watchdog timer.

A type-one reset changes settable configuration parameters back to their factory default values and does a type-zero reset. The weather station has no configurable settings, so both resets are identical.

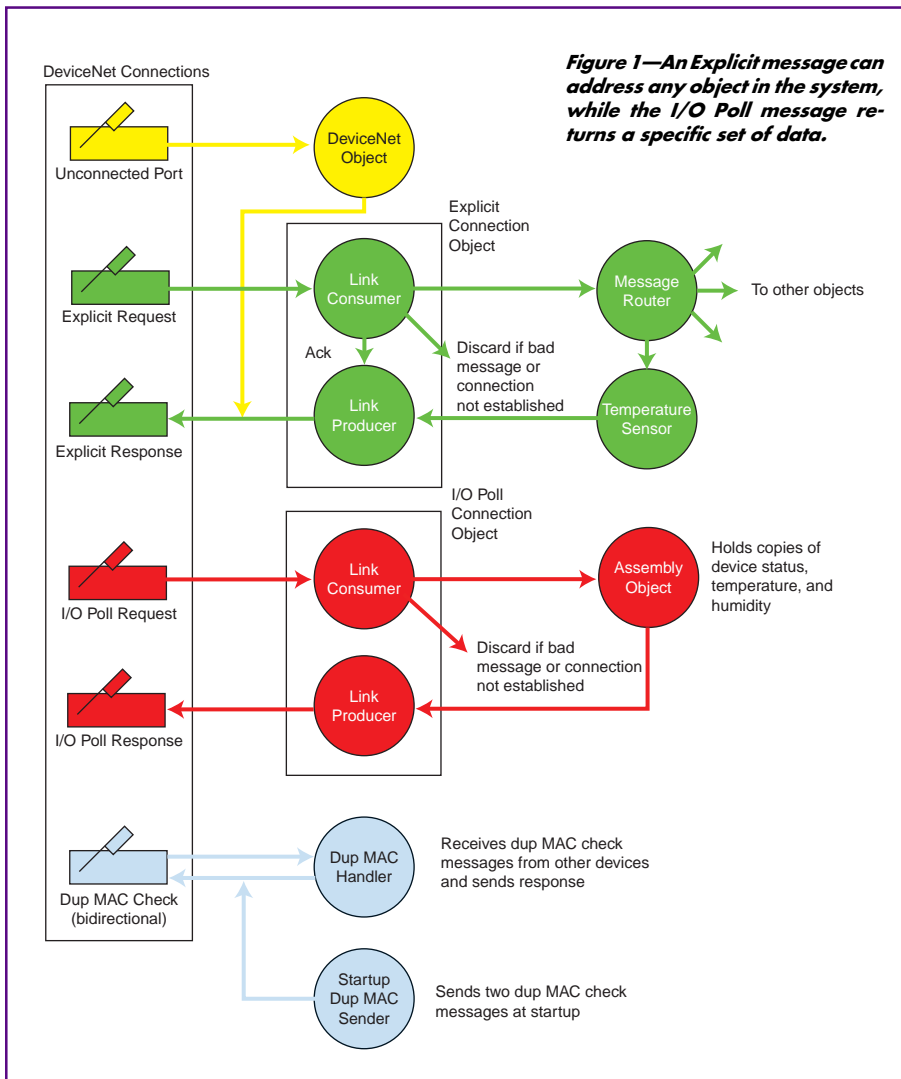
FRAGMENTED MESSAGES

The weather station's I/O Poll response is just three bytes—one byte each for device status, temperature, and humidity.

If I used floating point or added more sensors, the CAN message limit of 8 bytes would quickly be exceeded. I'd need to send the data in two or more fragments.

I/O message fragments are like normal messages except the first byte provides a fragment flag and a fragment count. That leaves seven bytes for data.

For maximum speed, I/O message fragments are sent back-to-back with no acknowledge (ack) message from the master other than the CAN level acknowledge bit.



Fragmenting an Explicit message is more complex. You send a fragment, wait for an ack message, and send the next fragment. If the ack takes too long, resend the fragment. If you time out again, give up trying to send the message.

Many error cases can arise, like getting an ack from the master with a fragment number different from what you sent, getting a message that's not an ack while you're sending fragments, and so on.

The weather station is capable of sending and receiving fragmented Explicit messages. Its serial number and product name are long enough to require it.

Fragmented messages are a big part of DeviceNet conformance testing. My

program managed to pass a self-inflicted test using the ODVA conformance software. This software generates every conceivable bogus response and breaks all but the best code. You like challenges, right?

thing from DeviceNet power. The voltage varies between 11 and 25 VDC, so use a wide input-range DC-to-DC converter.

DeviceNet also needs a miswiring-protection circuit, which lets you mix up

GETTING PHYSICAL

DeviceNet requires you to keep the network data and power isolated from green-wire ground by 1 MΩ or greater. If anything can reference your circuit to green-wire ground (e.g., an RS-232 port), you must optoisolate the network.

My PC/104 DeviceNet interface is shown in Figure 3. The weather station is isolated from ground and has no ports other than DeviceNet, so I didn't need optoisolators.

Power consumption is 5 W, so I powered the whole

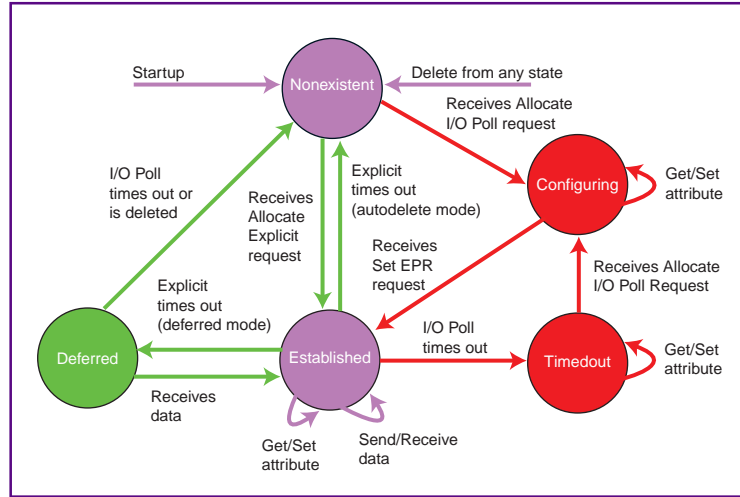


Figure 2—This state transition diagram for connection objects shows the events that cause the connection object to change state. Explicit connection states and events are shown in green, I/O Poll in red, and shared in violet.

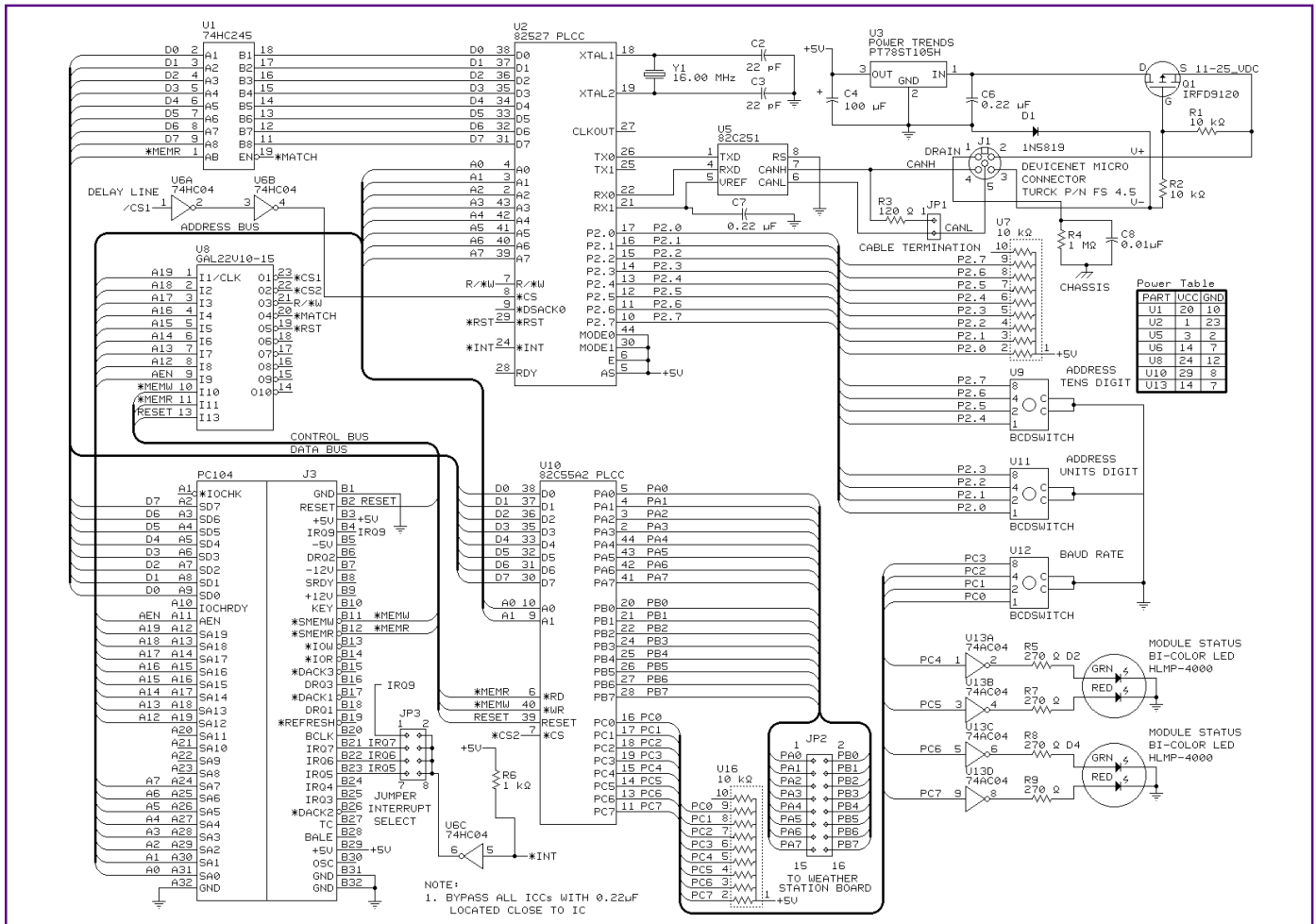


Figure 3—A simple eight-bit interface puts the Intel 82527 CAN controller on the PC/104 bus. The 24-V DeviceNet power is dropped to 5 V by U3 and then powers the entire weather station. Transistor Q1 protects against miswired network power. The PAL source code is available via the Circuit Cellar web site.

the network connections in any possible way without frying your device or the network. The DeviceNet specification includes a circuit for this. The Philips 82C251 CAN transceiver has ESD protection and line protection up to 40 V continuous.

DeviceNet is fairly specific in its interface guidelines. I used two BCD rotary switches to set MAC ID and one more for data rate. I also went with the recommended bicolor LED for module and network status.

Of the three network-connector choices, I used the circular micro style. It has five pins—two for differential data, two for power, and one for the drain wire.

The data lines are referenced to power V-, so your CAN transceiver must also be referenced to this to prevent exceeding its common-mode voltage range.

APPLYING DeviceNet

In addition to all of the objects for DeviceNet, you need code for your application. The weather station is simple enough that I just extended the Analog Input Point object. It reads the ADC and computes sensor values.

If you have separate application objects, you must link them with the Identity object. It keeps track of device status and does resets. A standard object used for SEMI-compliant devices, the S-Device Supervisor, is designed to do this.

This article is mainly about DeviceNet, but the weather station's details are on the Circuit Cellar web site. I hope to add sensors for barometric pressure and wind speed and build units for other locations.

My network master is a '486 with a DeviceNet card from Softing GmbH. It comes with a library that makes it easy to use. Also check out National Instruments' card which works with LabVIEW and CVI.

Aside from being a fun combination of real-time software and hardware, this project shows how straightforward it is to program a DeviceNet interface in C++. With a little care, C++ can provide good response times and reasonable program size. EPC

Jim Brady has designed embedded systems for 15 years. You may reach him at jimbrady@ix.netcom.com.

SOFTWARE

Complete source code and schematics for this article are available via the Circuit Cellar

Listing 2—Here's the Analog Input Point class for the temperature and humidity sensors. The explicit message handler allows the master to get any of the three attributes. They are read-only, so the Set Attribute service is not supported.

```
class ANALOG_INPUT_POINT{
private:
    UCHAR value;                // sensor value
    UCHAR data_type;           // data type of value
    BOOL status;               // alarm status
    static UINT class_revision; // revision of object
public:
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    void handle_explicit(UCHAR*, UCHAR*);
    ANALOG_INPUT_POINT() {value = 0; status = 0; data_type = 2;}
};

// Handle explicit request to Analog Input Point
void ANALOG_INPUT_POINT::handle_explicit(UCHAR request[],
    UCHAR response[])
{
    UINT service, attrib, error;
    service = request[1]; attrib = request[4]; error = 0;
    memset(response, 0, BUFSIZE); // clear response buffer
    switch(service){
        case GET_REQUEST:
            switch(attrib){ // return requested attribute
                case 3: // value
                    response[0] = request[0] & NON_FRAGMENTED;
                    response[1] = service | SUCCESS_RESPONSE;
                    response[2] = value;
                    response[LENGTH] = 3;
                    break;

                case 4: // status
                    response[0] = request[0] & NON_FRAGMENTED;
                    response[1] = service | SUCCESS_RESPONSE;
                    response[2] = (UCHAR)status;
                    response[LENGTH] = 3;
                    break;

                case 8: // data type
                    response[0] = request[0] & NON_FRAGMENTED;
                    response[1] = service | SUCCESS_RESPONSE;
                    response[2] = data_type;
                    response[LENGTH] = 3;
                    break;
                default: error = ATTRIB_NOT_SUPPORTED; break;
            }
            break;
        default: error = SERVICE_NOT_SUPPORTED; break;
    }
    if (error) // return error response{
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}
```

SOURCES

DeviceNet Information

Open DeviceNet Vendor Assn., Inc.
 (954) 340-5412
 Fax: (954) 340-5413
 www.odva.org

SBC1386

Micro/sys, Inc.
 (818) 244-4600
 Fax: (818) 244-4246
 www.embeddedsys.com

Digi-Key
 (218) 681-6674

Fax: (218) 681-3380
 www.digkey.com

DeviceNet cards

National Instruments, Inc.
 (512) 794-0100
 Fax: (512) 794-8411
 www.natinst.com

Softing GmbH
 ICT, Inc.
 (978) 557-5882
 Fax: (987) 557-5884
 www.softing.com

Real-Time PC

Ingo Cyliax

The Need for Speed

RTOS and PC/104

When it comes to speed, more is better—except when you're considering price. Ingo's ideas about speeding up transfers in and out of PC/104-based DES engines give us a less expensive product with the same performance.

Recently, our company went to the Design Automation Conference in San Francisco, where we demoed verified FPGA cores we derived using our EDA software. These cores were implementations of the Data Encryption Standards (DES), and this month, I want you to see how it worked.

We used a 486DX4-100 SBC from Versallogic to host our PC/104-based FPGA board. Photo 1 shows the setup.

A stand-alone web server I wrote in TCL was used to implement a simple encryption/decryption experiment. After the user input some plain text and a password on a web page (see Photo 2a), the web server loaded the DES encryption core into the FPGA board and encrypted the data into ciphertext (see Photo 2b).

Next, the user specified the same password to decrypt the message. The web server loaded the DES decryption core into the

FPGA board and decrypted the message to display on the web page shown in Photo 2c.

When I started testing this demo, I noticed the FPGA-based DES implementation was much faster than what it took to transfer data into it. This was kind of unnerving because I was using an oscilloscope to see how fast it was running. I could see the I/O operations that load the device and

read data from it, but I had the time base of the scope set too slow to even see the blip of activity while the hardware computed the DES.

At first, I thought something wasn't working, and I started to worry. Strangely, the data I was reading back from the board was encrypted correctly.

Although I knew that a hardware-based DES would be much faster than a software implementation, I didn't realize it spent so much time idling, waiting for the processor to poke the data into it.

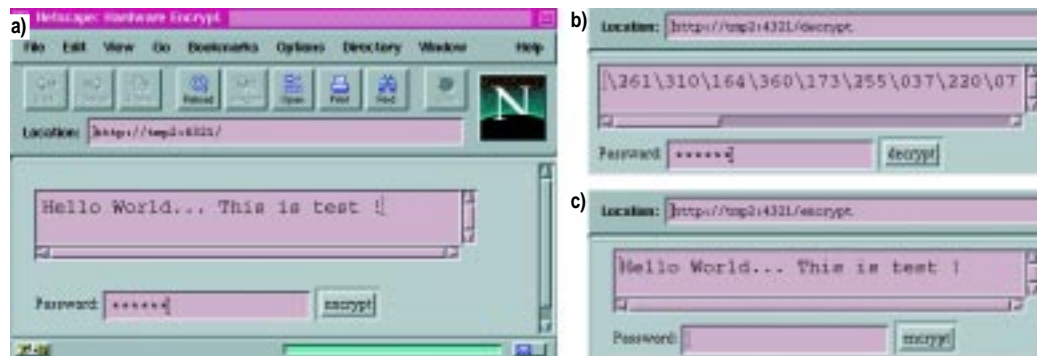
Sometimes it's hard to relate to how fast things really are. You need 10–50 μ s to transfer eight bytes of data on PC/104, but encrypting a text block only requires 0.5 μ s.

The transfer speed prompted me to look at PC/104 in detail and find a way to speed up the transfers in and out of my PC/104-based DES engine.



Photo 1—Our demo system consists of a CPU module on the bottom and an NE2000 Ethernet card module and a PF2000 module on top.

Photo 2— These screen shots show what the demo software looks like if you use a web browser on a regular PC. The user enters the plain text message and a password (a), which is then encrypted (b). Finally, the same password decrypts the message (c).



CLOCKING IN

PC/104 uses ISA-bus signals and timing. I thought this was going to be easy, but it wasn't. ISA bus, or Industry Standard Architecture bus, is a standardized interpretation of the original IBM-AT bus. It was also standardized by IEEE as P996.

These definitions mainly vary by the clock speed they recommend. The original PC used a 4.77-MHz clock, while the IBM AT went up to 8 MHz. Some AT clones even used 10–12-MHz bus clocks.

ISA recommends a clock of 8 MHz, and P996 specifies a clock of 8.33 MHz, which matches what my board is using. But, it gets worse.

On i386 and i486 motherboards, the ISA-bus clock is derived from the motherboard chip set, rather than the CPU clock. Usually you do this in the CMOS setup, under Advanced Setup. Look for an item called AT Bus Clock or Bus Speed.

Typically, the bus clock is some fraction of the CPU clock. So, a 33-MHz CPU clock needs to be divided by four to get a bus clock of about 8.33 MHz. By the way, hardware failures are often misdiagnosed because the ISA-bus clock is set too fast for one of the cards in the system.

OK, since PC/104 assumes 8.33-MHz clocks and my system uses 8.33 MHz, we'll go with that.

Bus-Cycle Type	States (read/write)
8-bit memory access	4/4
16-bit memory access	2/2
8-bit I/O access	5/5
16-bit I/O access	5/5
8-bit DMA access I/O-Mem	7/7
16-bit DMA access I/O-Mem	5/5
0WS	2/2

Table 1—Here's a summary of various bus cycles for PC/104 and ISA bus. Notice that 16-bit memory accesses are essentially performed at zero wait-state speed.

How do we relate the clock speed to what happens on the bus? ISA bus has several kinds of bus cycles, and each mode has some default number of states or clock cycles it takes to perform it.

In addition to the number of states needed to implement a specific bus cycle, there's a recovery time. That's the time between bus accesses, after one ends but before the next access can happen. The recovery time varies from implementation to implementation. Some motherboards let you set them in the CMOS setup.

Since we're dealing with I/O cards, I'm only going to look at I/O accesses. Each access mode has some default number of states it takes to perform the access, but there are two ways to modify the timing.

We can add wait states for slow devices or disable the default wait states and force the bus to do transfers in zero-wait-state mode. This latter option is kind of a misnomer because zero wait states are still implemented as two bus states. I experimented with my setup and took some measurements, which I'll tell you more about in a bit.

TIMING IS OF THE ESSENCE

Table 1 shows the state times for DMA transfers. Although any device can do DMA, it's slower on an ISA bus, especially with 8-bit accesses.

Also, ISA-bus DMA is hard to use. The program that wants to set up a transfer needs to initialize the DMA controller and then tell the I/O device to start the transfer.

So, a certain amount of processing is involved just to set up a transfer. It's not worth it unless the transfer is large enough to make the setup negligent in comparison.

And that brings me to the next limitation of ISA-bus DMA. You can only transfer within 64-KB segments. Unless the beginning memory address of a transfer

lies at the beginning of a 64-KB segment, the amount of data you can transfer is less than 64 KB. And when the transfer is done, the processor must be interrupted and restarted to do another transfer.

For a floppy, this might be a significant amount of data, since it transfers at low-data rates anyway and won't generate a high interrupt rate for large transfers. For multimedia, networking, or disk transfers, though, 64 KB isn't much. That's probably why the floppy disk is one of the few devices still using ISA-bus DMA.

The benefits of ISA-bus DMA aren't all that great, so most devices use polled I/O to transfer data. For polled I/O, you transfer the data to and from a data register implemented in the I/O register space.

You may think this technique is inefficient because the processor gets tied up executing I/O instructions and is forced to wait for a relatively long I/O cycle. After all, Pentiums clip along at 300+ MHz now, and even a two-state ISA-bus access at 8.33 MHz seems to take forever.

But, that isn't the whole picture. Most 32-bit processors in embedded PCs today have pipelines and several execution units that are separate from the load/store units. So, while an I/O operation is going on, the rest of the core can still process other instructions in parallel.

Here's one technique that can help speed up I/O transfers. Rather than transferring one byte or word at a time with the normal `inp` and `outp` instructions, try a string-based instruction. This instruction transfers a programmable number of bytes or words between an I/O port and memory. The prime advantage is that you only need one instruction to transfer larger amounts of data.

Now that you know a little more about I/O mechanisms and PC/104, it's time to test some of these assumptions.

THE THREE R'S

Because FPGA boards are programmable, I decided to implement a simple ISA-bus I/O interface, which enabled me to program how the card responds to PC/104 I/O accesses. I already did most of this for the original DES interface, so it was pretty easy to expand my interface to include the extra control signals necessary.

The first thing I added was a wait-state generator. The ISA bus uses the signal IOCHRDY to indicate when a card is done or ready with the data transfer.

Normally, this signal is pulled to a high state. If a card doesn't use the signal, it remains in a high state and the CPU chip set assumes the default number of wait states for that bus transaction.

To use the IOCHRDY line, a device needs to implement an open-collector driver and pull the line low (as long as it's not ready). The CPU chip set samples the line after some default number of wait states and holds the bus access until the I/O card releases the line and it returns to the high state, indicating that it's ready.

Addr	What
0x272	control/status register (CSR)
0x273	data port

Table 2—There are two ports in the register address map. The status/control port (CSR) controls the board's response to bus accesses. The 8-bit data port connects to an 8-byte deep shift register inside the FPGA.

I implemented a programmable generator. When the bus-access cycle starts, a counter loads the wait-state value from a control register and counts to zero. While the counter is not in the zero state, it asserts a low on the IOCHRDY line, telling the CPU to add wait states.

When it reaches zero, it sets IOCHRDY high. This line also has a tristate buffer that's only enabled when the card is being addressed. Now I can program how long IOCHRDY is asserted on this card by loading a register. A zero indicates that I don't want to use IOCHRDY, and it stays high for the bus cycle.

The other signal I want to implement is OWS (a.k.a. ENDXFER or SRDY). When asserted, it tells the CPU to use a no-wait-state cycle to access the card. Remember, a no-wait-state access takes two states.

The OWS line on my card is implemented via a tristate buffer. When the card is selected and a flag in the control register is set, the card asserts a low on the OWS line, indicating that the card can do a zero-wait-state access.

Finally, even though my card is only 8 bit, I wanted to get a grip on the timings of 16-bit accesses. So, I used a tristate buffer to implement the IOCS16 signal in the same way as the OWS signal.

The IOCS16 signal indicates to the CPU that the card can support word access using `inpw` and `outpw`. If you tried to access an 8-bit card (one that does not assert the IOCS16 signal) with a word I/O instruction, it would perform two 8-bit cycles transparently to the program.

With a programmable IOCS16 signal, I can make the CPU think there really is a 16-bit card. My card then responds by only giving the CPU the contents of one 8-bit register. The other signals are undefined. This is fine for doing a read.

Table 2 shows how the register on my card is implemented. CSR is an 8-bit register that lets me program the IOCHRDY wait states and the function of the OWS

signal. Figure 1 shows how these functions are mapped to the register.

The data register is a port into an 8-byte register. Reading and writing the register needs to be performed in eight cycles. The 8-byte register is a holdover from the DES implementation, which needs two 8-byte registers to hold the plain-text value and a key.

My ISA-bus module also implements the second 8-byte register, which is switched with one of the bits in the CSR. Figure 1 shows the register bit.

To test this interface, I wrote a program that enables me to set my interface modes, write patterns to the 8-byte registers, and read them back using the I/O string-transfer instructions. If the data read back matches the written data, it goes on to the next pattern. Using the string instruction for testing should represent the worse case because it reads and writes the data.

To handle timing, I wrote another program that reads and writes 8-bit values into the card by trying a variety of I/O-instruction mixes. To ensure that we get some good measurements, I ran one million accesses for each mode and measured the time it takes to run. The modes I ran are:

- read single byte
- read bytes as string
- write single bytes
- write bytes as string
- interleave reading and writing a byte

For 16-bit mode accesses, I only tested reading the card, since the card is only an 8-bit card faking 16-bit cycles. Writing to my card alters the CSR, but reading it is OK. I ran the same number of access cycles to get a measurement. For 16-bit mode, we have:

- read single word
- read words as string

272	0	0	IO16	OWS	X/K	Wait
273	Data					

Figure 1—In the register map for the PC/104 card, the lower bits (Wait) encode the number of states to assert the ICHRDY signal. If our card is being accessed, the OWS and I/O16 bits control whether the OWS or the IOCS16 signal gets asserted. The X/K bit selects whether the data port points to the X or K register.

I ran a series of tests with different wait-state combinations and OWS access modes. The results are illustrated in Figure 2.

The x-axis shows the wait states, while 0 indicates asserting the OWS signal and 1–8 indicate initializing the wait-state counter with the value of 0–7.

The y-axis shows the number of operations per second, which vary from 472 kop/s (kilo-operations per second) to 893 kop/s for the system I tested with this board and software.

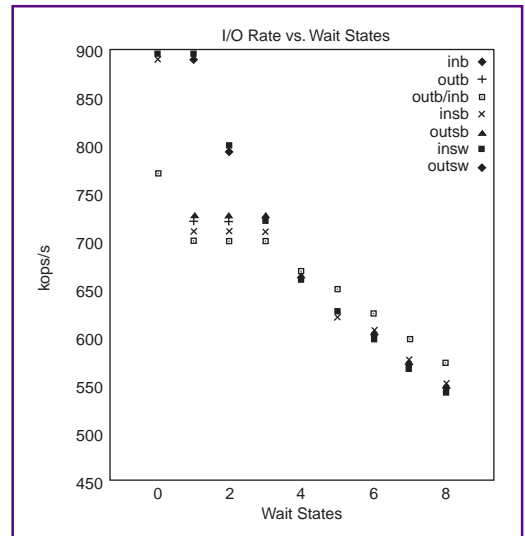
The single instructions (inb/outb) perform much like the string instructions

(insb/outsb). This similarity is due to the instruction cache of the processor (i.e., the loop doing single-transfer instructions is contained within the instruction cache), so there is little penalty to executing a bunch of them.

Also note that the 16-bit transfer modes do not implement default wait-state modes. That is, the OWS mode has the same timing as not asserting IOCHRDY.

In 8-bit mode, OWS is the fastest. But, selecting 0–3 for the IOCHRDY wait states has no effect on timing, which implies that

Figure 2—This plot shows how the different access modes perform. In particular, note that the single instructions (inb/outb) perform about the same as the string instructions (insb/outsb).



this motherboard always inserts three wait states in 8-bit mode unless OWS is selected.

I only ran the test with one CPU board, so I'm likely to get different numbers for other boards. The biggest difference will probably be the amount of recovery time implemented in the chip set. Of course, some boards will be able to change the bus clock, which will also affect the timing, but then it won't be standard anymore.

CATCH OF THE DAY

Now I know how to speed up transfers to our DES engine. First I'd use the OWS signal to force the CPU to use the fastest access mode possible. This technique improves the access time from around 700 kop/s to close to 900 kop/s.

Also, I can double the throughput to the DES engine via 16-bit transfers. Instead of only getting close to 900 kbps, I can get almost 1.8-Mbps throughput.

While I can more than double the original throughput to my board, this implementation is still I/O bound. Consider that the DES engine takes 500 ns to complete one encryption per 8-byte block.

If the key stays the same, I have to provide 8 bytes of data for the plain text and read 8 bytes of data back from the card. That takes 8 μ s using 16-bit transfers and 16 μ s in 8-bit mode.

Although this application is severely I/O bound, I now have some quantitative ideas about what I need. One solution is to find a faster bus, like PCI. At 500 ns, the DES core will need about 32 MBps of bandwidth to stream data.

Luckily, speed wasn't an issue for the demo. Our main task was showing that we could develop working FPGA cores, so I could switch to the slowest (and cheapest) FPGA speed grade for this implementation. Slower FPGAs are up to three times less expensive than the speed grade I was

using, and the DES would run at half the speed (1 μ s). Finding out something can be made cheaper is always nice.

In my case, it would have made the demo a bit cheaper, resulting in a nice fish dinner after the show. And of course, having a less expensive product with the same performance can mean higher profits or being more competitive. 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

- E. Solari, *ISA & EISA: Theory and Operation*, Annabooks, San Diego, CA, 1992.
- L.C. Eggbrecht, *Interfacing to the IBM Personal Computer*, SAMS, Carmel, IN, 1990.

SOURCES

PF2000 and Verified DES FPGA cores

Derivation Systems, Inc.
(760) 431-1400
Fax: (760) 431-1400
www.derivation.com

VSBC-1 and NE2000 modules

Versallogic Corp.
(800) 824-3163
(541) 485-8575
Fax: (541) 485-5712
www.versallogic.com

FPGAs used on PF2000

Xilinx Corp.
(800) 624-4782
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com

Applied PCs

Fred Eady

RF and Micros

Part 2: A Low-Power System

Fred's taken to the airwaves again. Using a DVP transmitter/receiver pair, he shows us how RF can be employed where wires previously dominated—with the amazing potential to touch 256 printers over a single RF link.

We live in the land of gadgets. Our world depends on the electronic goodies that spew from electronic minds like yours. Only the far reaches of humanity in the deepest tropical rain forests have seen little, if any, of this type of technology.

Again, last month, I was lucky enough to get some TV time. I was watching TLC (The Learning Channel, of course) and stumbled on this documentary about a civilization that exists in the rain forests of some far off and remote piece of one of the southernmost continents.

Anyway, small monkeys are a staple in the diet of these indigenous people. They like grilled tarantula, too, but that's another story.

What I'm getting at is that the men and their sons go about the hunt every now and then. Sorta like you and me going on the grocery-store trail, the difference being that at the meat

counter, the rain-forest dude uses a very, very long blowgun that propels a poisonous dart into the chosen primate. You know the rest.

In comparison, you and I swipe our debit card and realize the same result. Maybe not monkey meat, but meat, if that's your persuasion.

My point? One man's high technology is another man's blowgun. The common-

ality is that both the blowgun and the high-tech card reader will, at various times in their useful life, need maintenance or maybe even replacement.

Again, our topic du jour is low-power RF the embedded way. So, let's lay the foundation for an RF/embedded application using a couple of American Advantech Corporation PCM-4862s, a touchscreen monitor powered by MicroTouch, some ROM-DOS from Datalight, a thermal printer from Axiohm, and some Holtek-enhanced RF magic from DVP.

We're going to use a touchscreen as a source of input to key a transmitter that sends a message to one or more remote thermal printers.

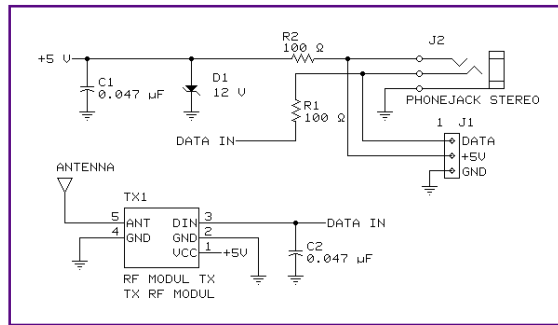
EMBEDDED INGREDIENTS

American Advantech produces a number of embedded computer systems, and I just happen to have the PCM-4862



Photo 1—The SSD is contained in the three 29C040s right above the BIOS EPROM.

Figure 1—
All of the
magic is inside
the DVP trans-
mitter module. It's up to
a lonely zener to help
out on the power end.



in my possession. My PCM-4862, which you see in Photo 1, runs at 100 MHz with 16 MB of RAM. I've used this board in previous applications, and the features it provides are perfect for this job, too.

The PCM-4862 is an all-in-one single-board '486 computer with onboard Ethernet, SVGA video, and solid-state disk. The embedded RF app I want to tell you about will use all of the aforementioned features. As well, it has a couple of RS-232 serial ports, a multimode parallel port, an enhanced IDE hard-disk drive interface, and a built-in floppy drive port.

Although I have my daily sessions with Counselor Troy, I haven't fully developed my telepathic abilities, and so I still need a mouse and keyboard interface for embedded systems I come into contact with.

I've also tried Scotty's "talk to the mouse" technique and still find I must place the mouse on a pad and physically move the little ball to effect cursor movement. I've got to have a talk with Jean-Luc about more time on the ship.

Meanwhile, I'll employ the services of the onboard keyboard and mouse interfaces for the development phase. I'll also need the serial and parallel ports to manipulate the DVP RF modules.

I probably won't use the IDE logic this time around. Thanks to ROM-DOS, I can fit the code plus DOS onto the SSD.

The first order of business is to install a bootable image of ROM-DOS on the SSD. Normally, the standard 3.5" floppy drive is designated A and the first hard disk is drive C. On the PCM-4862, drive A is indeed a floppy, while drive C is a trio of Atmel 29C040 512K x 8 flash-memory devices (this is sans a spinning disk or virtual disk that would normally take a C seat).

With the addition of a single jumper on the PCM-4862, A becomes flash and C goes the way of the bit bucket. After installing the jumper, I moved all of the necessary files from the ROM-DOS floppy

to the SSD and performed a SYS C. The SYS command put the necessary boot files out onto the flash.

A POR (power-on reset) was performed, and the PCM-4862 looked for boot devices and found the flash as drive A. A ROM-DOS 6.22 banner appeared followed by the familiar A:\> prompt.

Hey, a smokeless beginning. That's always good.

For those of you unfamiliar with ROM-DOS, it's essentially the embedded version of standard Bill DOS. To keep ROM-DOS's kernel below the 48-KB mark, it was compiled with '186 instructions rather than with true 8088 compatibility.

Since most "XT-compatible" machines on the market today are really '186 compatible, this situation is no big deal. Actually, it's a boost to 80xx and 8048x embedded developers who still depend on good old DOS.

All the features of Bill's DOS 6.x kernel (with the exception of the compression MRC1 interface) are built into ROM-DOS 6. This means the [MENU] commands in CONFIG.SYS work as expected. The

standard Billy DOS interrupt interface is there, and all the internal DOS structures match those of DOS 6.x.

Datalight's ROM-DOS also contains many features geared specifically toward the embedded developer. ROM-DOS is smaller in RAM than Billy DOS, and more importantly, it takes up about one half the ROM space that Bill's DOS does.

CONFIG.SYS processing can be reduced to DOS 5 level, DOS 3 level, or none at all for greater space savings. ROM-DOS is not only able to boot via ROM, hard, or floppy disk, but device drivers can be added to the ROM-DOS kernel, enabling booting off of any kind of disk. Bill's DOS is good, but if space is at a premium, ROM-DOS is a perfect substitute.

Remember Interlnk? Well, ROM-DOS does that, too. Datalight's ROM-DOS ships with a disk driver that can access a disk on a remote system via the serial port.

REMDISK.EXE and REMSERV.EXE are the client and server ends of what you might call a mini-network. REMSERV is run on the server end (the end that shares a drive), and REMDISK is run on the client end (which will gain a new drive).

Another addition is the small REMQUIT utility, which enables a user on the REMQUIT end to terminate execution of REMSERV on the opposite end of the link. There's a lot more to ROM-DOS than I need to tell you about right now, but let it be known that if you need to have a standard desktop DOS function in an embedded way, it most likely can be done with ROM-DOS.

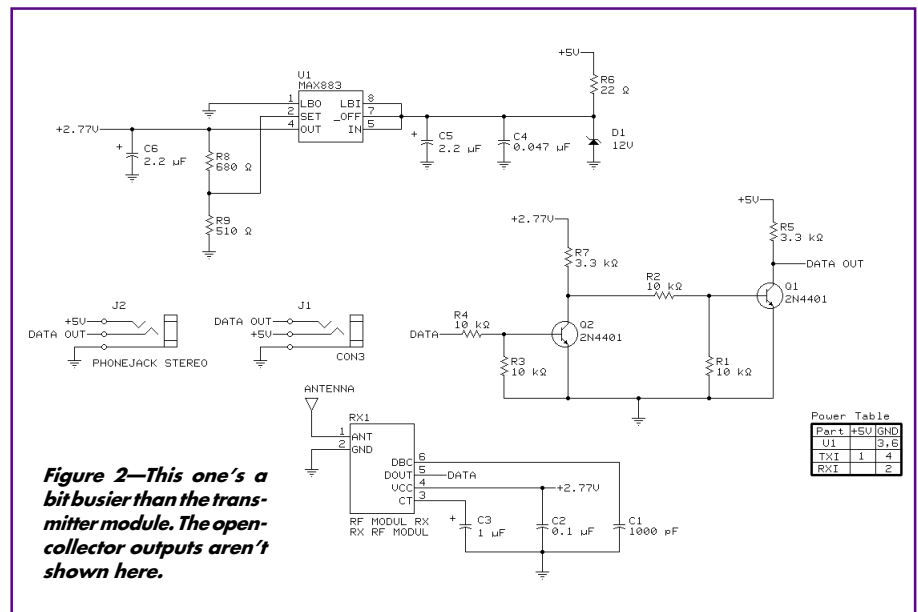


Figure 2—This one's a bit busier than the transmitter module. The open-collector outputs aren't shown here.

Part	+5V	GND
U1	1	2, 3
TX1	1	4
RX1	1	2

Now we've established a baseline. DOS is in the house, and so is anything else that can run under its 16-bit influences. Again, I'm in a no-C zone when it comes to software for this job.

As you know, I like C. It's disciplined. It's fast. It's universal. So what? I chose PowerBASIC for this project. It's disciplined. It's fast. It compiles. In short, PowerBASIC is much like C.

Ever try to manipulate binary with standard BASIC? Not. PB has bit-banging capability built in. This is due to the tight coupling of the PB command set to pure assembly routines.

I know what you're thinking, and you're right. Like C, you can do inline assembly with PB. There's even a PB DLL compiler that does DLLs for every language that can use them. The code is tight and fast.

The only problem I've encountered with PB is that sometimes the routines aren't very well behaved when interfacing with legacy applications written in native C.

This is good and bad. On the good side, it forces you to rethink the logic of the coded solution to your problem. Unfortunately, you also have to rethink the logic associated with the coded solution to your problem.

Again, there's nothing wrong with C, I just choose not to use it here. If you wanted old standards, you wouldn't be reading *INK*. Let's move on.

We've defined the capabilities of our hardware from the embedded aspect, and we've (I've) decided what language we will speak in this embedded country. Now, let's take a close look at the main ingredient of our embedded application—the DVP RF modules.

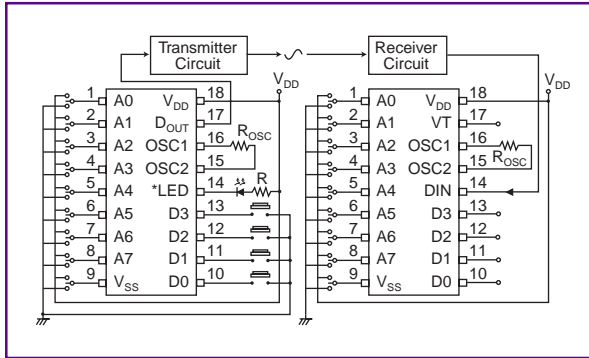


Figure 3—Oscillator frequency is set by a single 5% resistor.

A CUP OF RF

I have the raw DVP RF modules, but why waste good engineering? My DVP demo boards include the original RF modules, an integrated etched resonant loop antenna, and time-saving Holtek encoders/decoders. The guys and gals at DVP even take the outputs of the decoders to open collector mode, so you can interface to just about anything electronic on this planet.

No board design, no soldering (except what you want to), no guesswork. Did you say quicker time to market?

Oh, what's that? You don't do RF, but this setup may be your solution to that RF project you were assigned to. Hmm.... On my side of that equation, it equates to quicker time to the page because I didn't have to dig and design the RF for this application. Goodness for all!

With the DVP development modules, the RF is a given. So, let's look at how we transmit our digital data and receive it over the ether.

Rats! No matter how hard I try, I can't make this difficult. Figure 1 is typical of what's on the transmitter side. It may not be rocket science, but if you're RF impaired, this is a moon shot. Figure 2 shows us the receiver module.

I could write code to effect the desired datastream, but again, why waste good engineering? Instead, all the hardware shown in Figure 1 is married to the on-board Holtek 60xx encoder/decoder pair.

Figure 3 is a representative layout of the Holtek hardware found on the demo boards. The Holtek encoder takes care of



Photo 2—This diagnostic layout can be whatever your application demands.

the building of the data packet by adding address and data information taken from the address and data logic inputs. I described this process in detail last time.

A PINCH OF MICROTUCH

I could have used a keypad, mouse, or keyboard to select the necessary information, but a touchscreen is a better choice. With a touchscreen, I can apply this application to many industrial uses where keyboards and mice are apt to be damaged or just plain in the way.

The MicroTouch-enhanced monitor is a simple SVGA tube that has been modified with a capacitive touchpanel and integral microcontroller complex. The MicroTouch microcontroller senses a touch on the CRT and converts it into x,y coordinates that are transmitted serially to the PCM-4862.

Instead of using the raw x,y data, a standard mouse driver is employed, thus turning each touch event into a mouse event. Using PowerBASIC's MS Mouse support in conjunction with a standard mouse driver enables me to simply draw the buttons on the screen and equate them to a particular x,y point.

The PCM-4862 serial port at 0x3F8 (COM 1) is used to interface to the MicroTouch microcontroller. Listing 1 consists of PB code snippets that deal with the MicroTouch interface, and Photo 2 is a shot of the virtual keyboard.

A DASH OF AXIOHM

The Axiohm is a serial thermal printer that operates at 9600 bps. It's typically used as a receipt printer in point-of-sale applications. I chose the Axiohm because it was small and quiet. There's already enough noise in the Circuit Cellar Florida Room as it is.

One of the neat features of this printer is that you load paper by simply opening the top cover and dropping in the roll. This printer can also drive a cash-drawer solenoid. Now that you've seen it here, when you go out to eat or buy some lumber, you'll undoubtedly see it again and again.

LET'S COOK

All the ingredients are on the counter—the Advantech PCM-4862, ROM-DOS, a DVP receiver and transmitter, a MicroTouch monitor, and an Axiohm printer.

The idea here is to show how RF can be used where wire normally dominates. A touch to the MicroTouch-enabled monitor is detected and passed via a serial connection to the PCM-4862.

Under the control of ROM-DOS, a PowerBASIC program decodes the touch event as a mouse event. A bit pattern corresponding to the area touched is placed on the Advantech parallel port.

Listing 1—This snippet of touchscreen code opens a COM port, checks the modem control signal status, and prints a test message.

```

case 13 to 18
  showbuttons
  showstatus
  noprtflg=0
  a=GetComAddress(2)
  port=2
  qbox 6,2,3,23,15,0
  qprint 7,3, "TEST RECEIPT PRINTER",15
  cleanup
  MsReleaseWait
  beep 1
  qbox 6,2,3,23,14,0
  qprint 7,3, "TEST RECEIPT PRINTER",9
  z=FREEFILE
  open "com2:9600,n,8,1" as z
  dtrstat=10
  dtr$="OK"
  t=DtrStatus(2)
  if t = 0 then
    dtrstat=4
    qprint 7,26, "<-- COM2 DID NOT ASSERT DTR",4
    noprtflg=1
  end if
  dsr$="OK"
  srstat=10
  s=DsrStatus(2)
  if s = 0 then
    dsrstat=4
  end if
  rtsstat=10
  rts$="OK"
  ctsstat=10
  cts$="OK"
  c=CtsStatus(2)
  if c = 0 then
    ctsstat=4
  end if
  cd$="OK"
  cdstat=10
  d=Carrier(2)
  if d = 0 then
    cdstat=4
  end if
  showstatus
  if noprtflg=1 goto NOPRT2
  for x=1 to 10
    print #z,"PRINT TEST FOR RECEIPT PRINTER"
  next x
  print #z,crlf$
  print #z,"RECEIPT PRINTER AT COM2 ADDRESS = ";hex$(a)
  print #z,"DTR CONTROL LINE STATUS ==> ";dtr$
  print #z,"DSR CONTROL LINE STATUS ==> ";dsr$
  print #z,"RTS CONTROL LINE STATUS ==> ";rts$
  print #z,"CTS CONTROL LINE STATUS ==> ";cts$
  print #z,"CD CONTROL LINE STATUS ==> ";cd$
  for x = 1 to 10
    print #z,crlf$
  next x
NOPRT2:
close

```

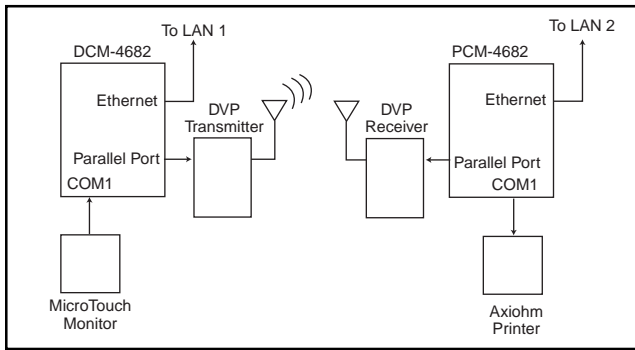


Figure 6—Look Ma, no wire!

Attached to the parallel port is a Holtek encoder (on the DVP transmitter demo board). The parallel port bit pattern sets address and data information to be transmitted. The transmitter is keyed by a low on the encoder, and the 12-bit data pattern is converted to RF energy.

At the receiving end, another PCM-4862 controls the printer. The receiving decoder feeds the transmitted bit pattern to the receiving PCM-4862 parallel port. The transmitted bit pattern is then decoded and translated into a message that is printed on the Axiohm thermal printer.

If all of that sounds simple, it is. Figure 6 schematically depicts all the parts that make up the application.

The power of this idea lies in the addressability of this encoder/decoder pair. Our application uses only 8 bits of the parallel port. Depending on how the address and data bits are allocated, this could limit how many printer complexes can be addressed over the RF link.

But by adding some multiplexing and latch glue, the entire address range can be accommodated. Up to 256 printers could be touched over a single RF link.

The Holtek parts take much of the code writing out of this application. By coding specific bit patterns for transmission, there's no limit to the possibilities.

The only requirement: follow the duty-cycle rules that apply to these DVP devices. Writing our own bit-pattern code lets us interface to the transmitter and receiver via the PCM-4862 serial ports, freeing up the parallel ports for other purposes.

In putting this application together, I assumed that each end of the RF link would embody massive embedded intelligence as found in the Advantech embedded PC. If the targeted printer is in a stand-alone situation, you can use a smaller embedded solution like a Microchip PIC to decode the RF transmission.

Once the RF energy does its work, the Ethernet capability of the Advantech PCM-4862 could be employed to move the data in a LAN environment. Instead of targeting a printer, you could target specific LAN segments. The Advantech PCM-4862 includes a set of disks that include Ethernet drivers for all of the popular operating systems.

DVP has taken the mystery out of incorporating RF into legacy applications. This time, it's DVP, not me, who has once again proven that it doesn't have to be complicated to be embedded.

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

PCM-4862

American Advantech Corp.
(408) 245-6678
Fax: (408) 245-8268
www.advantech-usa.com

60xx encoder/decoder

Holtek Microelectronics Inc.
+886 35-784888
Fax: +886 35-770879
www.holtek.com

RF modules

DVP, Inc.
(818) 541-9020
Fax: (818) 541-9423
www.dvp.com

Touchscreen Monitor

MicroTouch Systems, Inc.
(800) 642-7686
(508) 659-9000
Fax: (508) 659-9400
www.microtouch.com

ROM-DOS

Datalight
(360) 435-8086
Fax: (360) 435-0253
www.datalight.com

Serial thermal printer

Axiohm, Inc.
(612) 638-9856
Fax: (612) 638-0758
www.axiohm.com

DEPARTMENTS

68

MicroSeries

76

From the Bench

80

Silicon Update

Digital Processing in an Analog World

Basic Issues

Part
1
of
3

Conversion is always a hot topic, but David's not talking religion here. Instead, he wants to fill us in on the basics of A/D and D/A conversion, so that we'll be ready to delve into the deeper issues in the coming months.

MICRO SERIES

David Tweed



Signal processing in the digital world requires that analog signals be converted to discrete units in both a measurement dimension (voltage, current, temperature, etc.) and time.

The former is called quantization, and the latter is known as sampling. While these conversions can be analyzed independently by a mathematician, a real-world analog-to-digital converter deals with both simultaneously.

Similarly, when digital processing is complete, it's often necessary to convert the signal back to a continuous-measurement, continuous-time domain. That's the job of the digital-to-analog converter.

In this series, I discuss the fundamentals of A/D and D/A conversion. Part 1 covers the basics of reading and understanding specification sheets. Part 2 introduces the most important conversion technologies as well as their strengths and weaknesses, relative to the parameters I cover. In the final article, I delve into delta-sigma conversion and show you how to dither.

RANGE, RESOLUTION, AND ACCURACY

An ADC maps an analog measurement to a set of digital codes that can be conveniently manipulated. In the ideal case, this mapping is perfectly linear and can be drawn as a straight line on a graph. However, the fact that

the analog domain is continuous and usually unbounded while the digital side has a finite set of codes creates two ways in which the graph deviates from the straight line.

First of all, the converter's range has definite endpoints. All analog values outside this range simply map to the highest and lowest codes.

Also, each digital code represents a small range of analog values, creating a stepwise mapping from the analog to the digital domain, as Figure 1 shows.

Similarly, a DAC maps digital codes into analog values. Given the finite set of digital codes, there must also be a finite set of discrete analog values the converter can generate. For an ideal converter, these points lie along a straight line.

Figure 1 shows the transfer functions for an ideal three-bit ADC and the corresponding three-bit DAC. The dashed line represents the ideal straight-line transfer function that each unit tries to approximate.

As you see, the ADC divides the analog domain into a series of ranges and assigns one digital code to each range. The range on each end is open-ended, while the six intermediate ranges are the same fixed size.

The eight ranges are defined by seven decision levels or input values that cause the converter to switch from one

code to the next. In real applications, the end ranges are usually the same size as the intermediate ranges, and a single value is associated with each range that is the midpoint of the range. The size of each range is called the step size or quant (i.e., quantum or smallest perceivable change).

The corresponding three-bit DAC produces the single value associated with a range when given the digital code for that range. Together, the ADC and DAC form a system that has a stepwise linear transfer function.

The maximum difference (or error) between input and output occurs when the input value is very close to a decision level, and as you can see, the corresponding output value differs by half a quant one way or the other.

The values associated with the end ranges are known as full-scale values. In a unipolar converter, the lower end is usually zero and the upper end is a positive value (e.g., 5 V). In a bipolar converter, the end values are plus and minus the same value (e.g., ± 2.5 V). The converter's range is the difference between the end values (here, 5 V).

The number of bits in the digital code determines how many codes there are by a simple relation:

$$\text{codes} = 2^{\text{bits}}$$

This equation gives the number of input ranges for an ADC or the number of output values for a DAC. Since the input ranges are the same size, their size is given by:

$$\frac{\text{range of converter}}{2^{\text{bits}}}$$

Think of the quantized signal as the sum of the original signal and an error signal introduced by the quantizer, as shown in Figure 2. Horizontal dotted lines represent the decision levels.

This error signal varies in a complex way that depends on the input signal, so it is usually treated and analyzed as a noise source in the ADC.

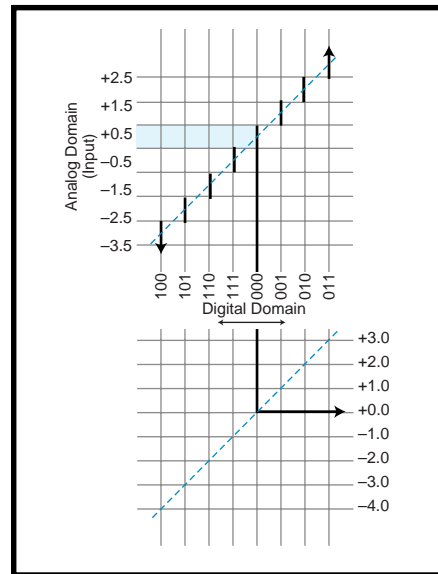


Figure 1—In an ideal world, converters have transfer functions that approximate straight lines. ADCs translate ranges of input values to discrete codes, and DACs translate those codes to specific output values.

However, to massage the noise signal you need to know how much noise is generated (amplitude) and what its spectral characteristics are (i.e., in audio systems, what does it sound like?).

If the peak amplitude of the error signal is limited to half a step size (in an ideal converter), how much power (loudness) does this represent? Assuming that the error signal is a sawtooth signal with a range equal to the step size, then its RMS value is:

$$\frac{\text{stepsize}}{\sqrt{12}}$$

which is about 0.289 times the step size.

The RMS value of a full-scale sine wave is:

$$\frac{\sqrt{2}}{2}$$

or about 0.707 times its peak value. For an n -bit converter, the ratio of the sine-wave RMS voltage to the error signal RMS voltage is given by:

$$\frac{0.707 \times \frac{\text{range of converter}}{2}}{0.289 \times \frac{\text{range of converter}}{2^n}}$$

The converter's range cancels out, so we are left with a simple expression that depends only on the number of bits:

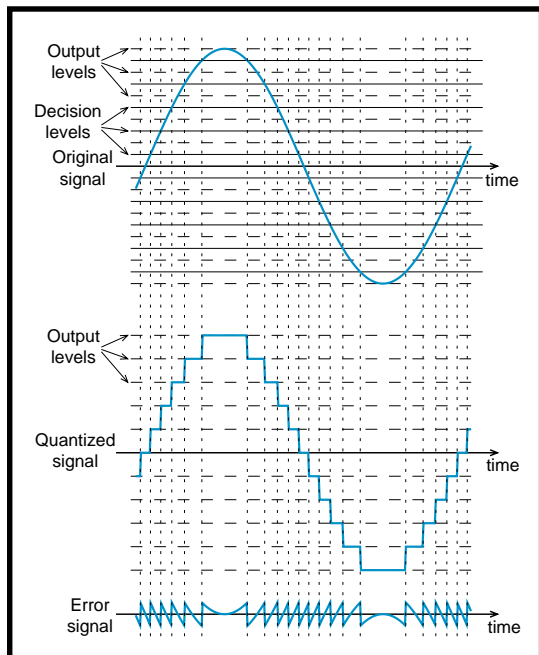


Figure 2—A quantized signal can be represented as the sum of the original signal and an error signal.

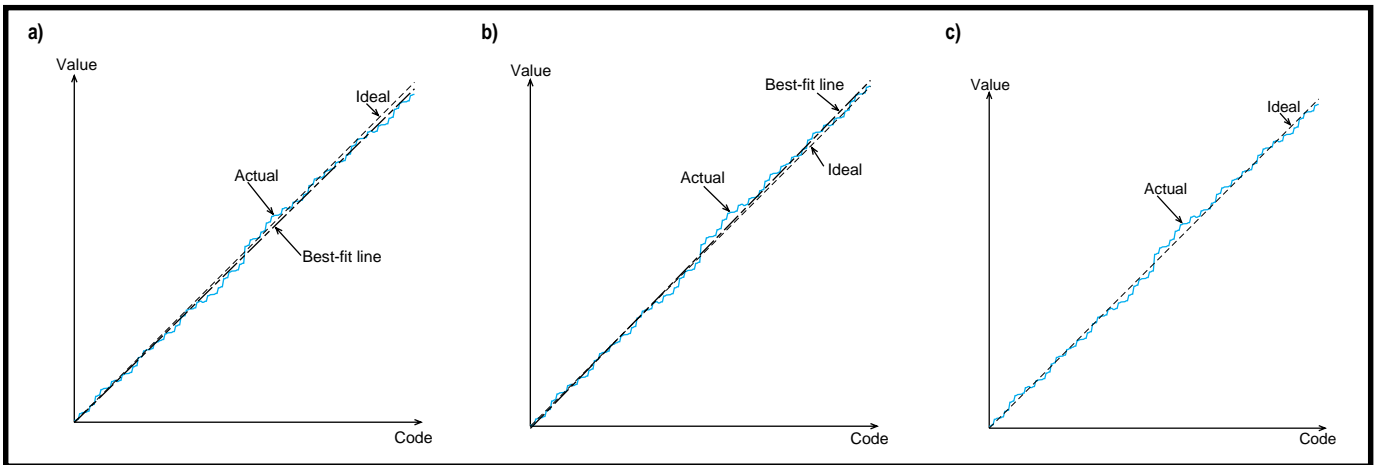


Figure 3a—A typical uncalibrated converter curve deviates from the ideal. However, you can approximate the ideal by calibrating the endpoints (b), a simple method which unfortunately has a larger peak error, or by calibrating the best-fit line (c), which yields the lowest RMS error.

$$\frac{0.707 \times 2^{-1}}{0.289 \times 2^{-n}} = 2.45 \times 2^{n-1}$$

$$= 1.225 \times 2^n$$

This range is usually given in decibels (e.g., a logarithmic scale), which works out to:

$$20 \times \log(1.225 \times 2^n) = 1.76 + 6.02 \times n \text{ dB}$$

This 3-bit converter has a signal-to-noise (S/N) power ratio of about 20 dB, but an ideal 16-bit converter should achieve 98 dB. Adding a bit of resolution increases the S/N ratio by about 6 dB.

Figure 2 shows how the characteristics of the error signal vary over the full cycle of the sine wave, which is often undesirable.

For example, in an audio converter, a low-frequency sine wave can noticeably modulate background noise, which is more annoying than a steady level. You can solve this problem by adding a randomizing signal (i.e., dithering it) before you quantize it.

The dither signal's characteristics are chosen to mask the effects of quantization without becoming objectionable itself. I'll discuss dither signals and their application more in Part 3.

LINEARITY

A real-world converter isn't going to have the perfect straight-line transfer characteristic of the ideal converter. Figure 3a shows some of the problems you may find in a transfer curve.

One way to get this converter working in an application would be to

adjust the curve's endpoints to the desired values, letting the intermediate points fall where they may (see Figure 3b). This way, it's easy to calibrate by examining two points on the curve.

You can also find the best-fit straight line for the converter and calibrate that to the desired range, as shown in Figure 3c. This approach gives the lowest overall RMS error if the full range of the converter is used, but unfortunately, you need to examine a large number of points on the curve to find the best-fit line.

In either case, the deviations from the straight-line characteristic (known as nonlinearity) distort the analog signal's digital representation. These distortions are the same as you'd find in a purely analog nonlinear system and are broadly classified as harmonic (waveform) distortions and intermodulation distortions (e.g., interactions among simultaneous signals).

Manufacturers have a number of ways to characterize their converters for distortion. One way, differential nonlinearity (DNL), is simply the variation of the step size for each digital code from the ideal theoretical value.

For an ADC, this is the difference between successive decision points, while for a DAC, it is the difference between successive output values. Figure 4a graphically depicts these results, placing the digital codes along the x-axis and the error associated with that code along the y-axis.

Integral nonlinearity (INL), which is just the transfer curve of the device,

is another way to characterize distortion. An integral curve (see Figure 4b) can be generated by integrating over the differential curve. Vendors usually show one curve or the other, depending on which characteristic of their device they want to emphasize.

Which is more important? It depends on your application. If absolute accuracy is crucial (e.g., measuring a voltage or current in an industrial process), the integral nonlinearity is more important.

But in an audio application, integral nonlinearity represents an overall gain error that is, for all practical purposes, irrelevant. Differential nonlinearity gives a better idea of the audible distortions the converter produces.

Another way to characterize an ADC is to feed in a full-scale sine-wave signal and make a mathematical histogram of the number of times the converter produces each code, as you see in Figure 4c.

An ideal converter produces a specific curve (related to arc sine) under this test, and examining the way a histogram deviates from the ideal is a quick way to find potential problems, including missing codes that the converter never produces.

Some ADCs have larger than average nonlinearities when a more significant bit changes state (e.g., going from 00111111 to 01000000). In some cases, the decision level implied by 00111111 is actually slightly higher than that of 01000000.

A slowly rising signal generates a code sequence that jumps directly

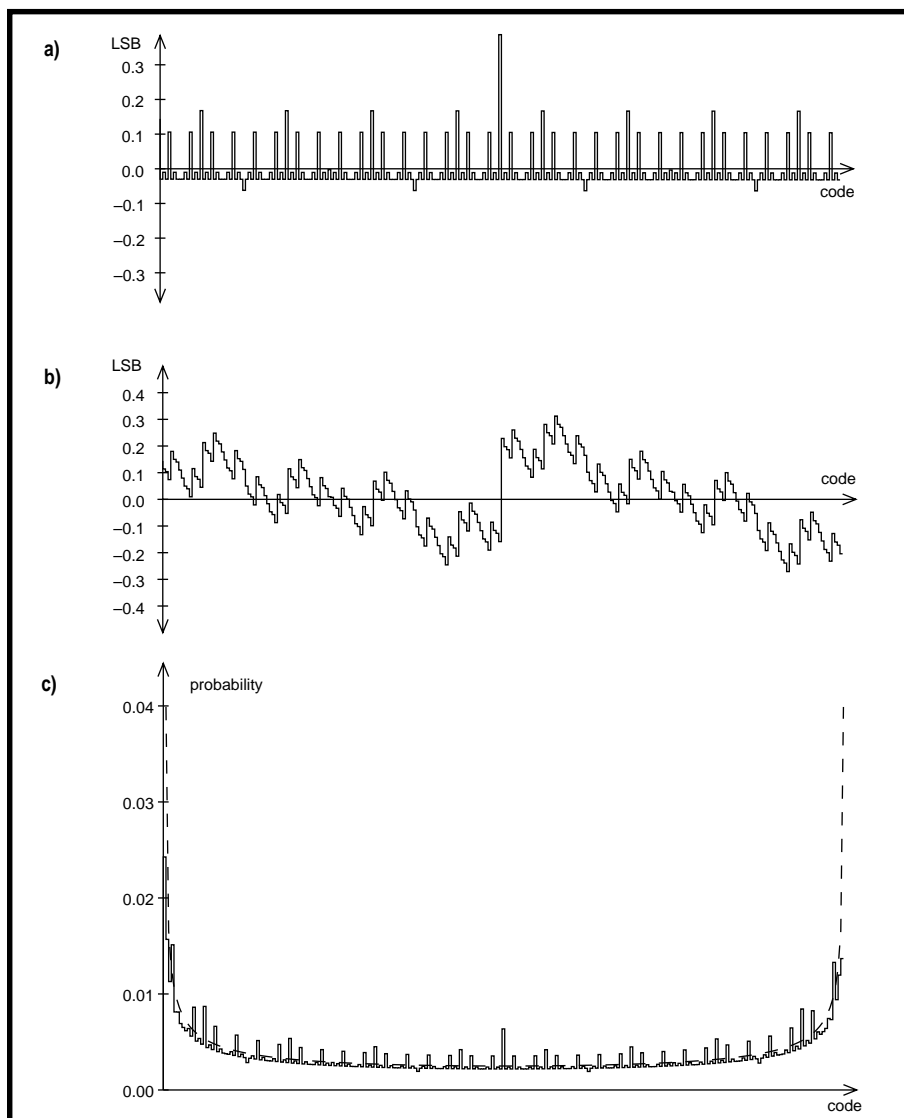


Figure 4a—The differential nonlinearity curve shows the step-to-step error, while the integral nonlinearity curve shows the cumulative error (b). c—The histogram is a way to measure DNL (dashed line shows ideal shape).

from 00111111 to 01000001, and the code 01000000 is said to be missing. While this miss doesn't cause large problems in terms of the overall error or noise level, it can indicate an underlying problem, so, manufacturers like to boast "No missing codes."

You can characterize the overall linearity of a converter by specifying the spur-free dynamic range (SFDR), which is measured by applying one or two full-scale sine waves to the converter and doing a spectral analysis (e.g., FFT) of the output.

The SFDR is the ratio between the peaks representing the original signal(s) and the highest peak of any of the harmonic or intermodulation distortion products. This performance measurement is most important in the high-

speed converters used in digital radios but can be important in other applications as well.

SAMPLE RATE AND BANDWIDTH

As mentioned, the codes in a DSP's memory represent points or discrete values along the measurement and time axes. Quantization gets us the first, and sampling gets us the second.

Shannon and Nyquist showed that a continuous-time band-limited signal can be perfectly represented by a set of discrete samples as long as the sample rate is greater than twice the bandwidth of the signal.

In many systems, the frequency band of interest includes DC, so it is often stated that the sample rate must be greater than twice the highest fre-

quency in the signal. However, you need to make the distinction since there are systems in which the bandwidth is considerably narrower than the frequencies present (e.g., the IF stage of a digital radio receiver).

If a system fails to meet the Nyquist criterion, then aliasing occurs, causing signals at frequencies that originated outside the Nyquist bandwidth to have the same effect as a phantom signal within the Nyquist bandwidth. Once this happens, it's impossible to separate the undesired signal from the desired signal (but see Gerard Fonte's article in this issue).

If the signal is not known to lie within the Nyquist bandwidth, it must be filtered in the continuous-time domain before sampling occurs. Let's look at two basic approaches to this.

You can place analog filters ahead of the ADC that attenuate out-of-band signals at or near the quantization step size. But if signals just within the Nyquist bandwidth are important to the application, you need high-order filters with steep skirts. These filters tend to be complicated (high parts count) and difficult to adjust, and they introduce undesirable phase shifts.

The second approach minimizes these effects by oversampling (i.e., using a much higher sample rate initially). This technique raises the Nyquist frequency, enabling you to use a much simpler analog filter. Digital-filtering techniques that are easier to control and that have better phase characteristics can reduce the bandwidth before resampling the signal at the desired final sample rate.

Delta-sigma-based converters take this to the extreme by using a one-bit converter at a very high sample rate to achieve the performance of a multibit conventional converter. I'll cover delta-sigma conversion in Part 3.

SAMPLE TIMING AND JITTER

Just as there can be small errors in the exact placement of decision points in an ADC's measurement domain or in the placement of output levels in a DAC, both kinds of converters can have small errors in the time the samples are taken or generated. This error is called timing jitter.

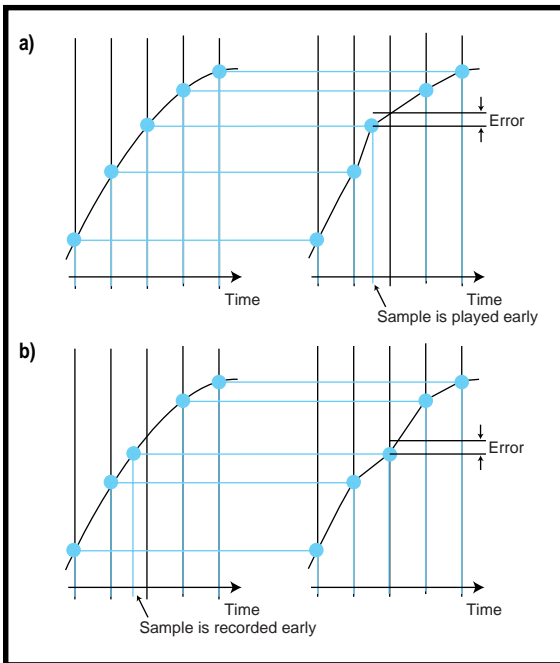


Figure 5a—A timing jitter in a DAC produces errors similar to quantization noise, and as you can see, a timing jitter in an ADC has much the same effect (**b**).

As Figure 5a shows, errors in the horizontal (time axis) placement of samples by a DAC produce errors in the resulting waveform that are similar to quantization noise. The magnitude of the error in the measurement domain is related to the size of the timing error by the slope of the signal. For a DC signal, timing jitter has no significance, but signals near the converter's bandwidth limit have a serious effect.

In the worst case, a converter will slew over its entire range from one sample to the next, so a timing error producing a one-quant error equals the sample period divided by the number of steps. For CD-quality audio (i.e., 16 bits at 44,100 samples per second), this works out to 346 ps (10^{-12} s), which is very tiny indeed.

A more typical example is a full-scale 1-kHz sine wave, which has a maximum slew rate of 4669 steps per sample. A 4.85-ns timing error produces a one-quant error here.

However, it's rare for even a relatively inexpensive crystal oscillator to have 0.5 ns of jitter. It's pretty safe to ignore most of the breathless hype you read about the horrible jitter problems of CD players.

Figure 5b shows how timing errors that affect when an ADC accepts its samples have the same effect when the samples are played back through a perfectly timed DAC.

CONVERSION SPEED

Except for flash converters, which do their work instantly, most ADC technologies need a finite amount of time to complete a conversion. Many assume the signal won't change significantly (more

than the step size) over that period.

However, some systems can't make that guarantee inherently. A sample-and-hold or a track-and-hold circuit is then placed in front of the converter that samples the signal in analog form and holds it during the conversion.

Figure 6a shows this kind of circuit. The input signal charges a low-leakage capacitor, whose voltage tracks the signal as long as the switch is closed. When conversion begins, the switch

opens and the capacitor holds the level of the input signal for that moment. A buffer amplifier with an high input impedance keeps the voltage from changing until the switch closes again.

When designing sample-and-hold circuits, also pay attention to:

- how long it takes for the output to match the input once the switch closes again
- whether there are any offsets between the input and output values
- how long the circuit can hold the desired value to the needed degree of accuracy (droop rate)

Sometimes the signal controlling the switch feeds through to the circuit output, making the offset with the switch closed different than with the switch open. It's impossible to completely eliminate these errors, but you minimize them as much as possible relative to the converter's resolution.

As Figure 6c shows, track-and-hold circuits also have applications in conjunction with DACs. Sometimes the output of a particular DAC has glitches at the moment a new sample comes along, rather than moving smoothly to the new value.

These glitches tend to be short relative to the sample period, and sometimes you can control them by simple low-pass filtering at the output of the DAC—the same filter that

eliminates the images of the output spectrum.

If the glitch energy (amplitude multiplied by duration) is high, the heavy filtering required would cause undesirable effects to the signal, so a track-and-hold circuit is used to disconnect the DAC from the output for the duration of the glitch.

ZERO-ORDER HOLD

The mathematics of sampling theory assumes that the samples have infinitesimal width. However, because an impulse function has a

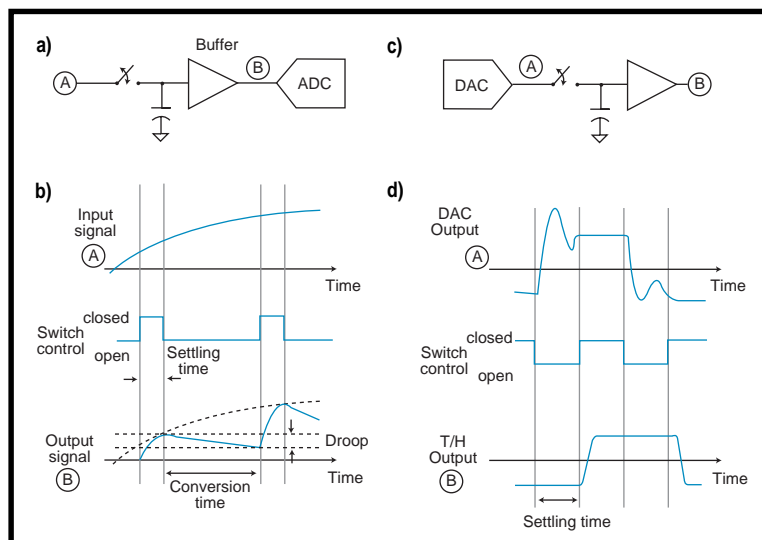


Figure 6a—A sample-and-hold circuit holds a steady value while the ADC completes a conversion. **b**—The settling time and the amount of droop during the conversion are important characteristics. **c**—A track-and-hold circuit can hide the output glitches of a DAC. **d**—However, the output signal is delayed by the length of the settling time.

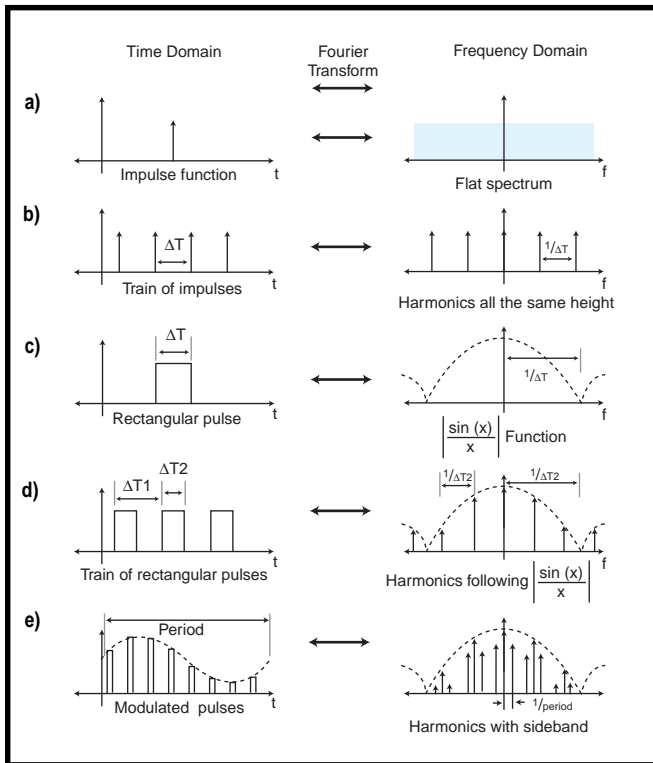


Figure 7—Fourier transform pairs show the relationship between a time-domain signal and its spectrum for a single impulse, which has a flat spectrum (a), an impulse train, which has a periodic spectrum (b), a rectangular pulse, which has a $\sin(x)/x$ spectrum (c), and a rectangular pulse train, which has a discrete version of the $\sin(x)/x$ spectrum (d). e—The output of a DAC is a modulated train of rectangular pulses.

flat frequency response (see Figure 7a), a train of such impulses has a spectrum that is a discrete version of a single pulse as illustrated in Figure 7b.

As Figure 7c shows, a rectangular pulse has a $\sin(x)/x$ spectrum. As the pulse narrows, the spectrum widens. In the limit, the pulse becomes the impulse function and the spectrum is infinitely wide or flat. Figure 7d presents a train of rectangular pulses with a discrete version of the same spectrum.

What does this have to do with a DAC? As a designer, your concern is what the converter does in the time between the instants defined by the samples. The math assumes the function is zero between those instants, but real-world converters do some kind of interpolation.

First-order interpolation draws a straight line between one sample and the next. Second-order interpolation use a quadratic function, in which higher orders of interpolation use higher orders of polynomial functions.

In fact, most converters hold a value until the next sample comes along. This technique is called zero-order interpolation, zero-order hold, or $\sin(x)/x$ correction.

Mathematically, you can treat the DAC output as a series of rectangular

pulses wide enough to completely fill the interval between the samples, as indicated in Figure 7e. Because $\Delta T1$ equals $\Delta T2$, the nulls of the $\sin(x)/x$ curve fall exactly on the harmonics of the sampling frequency, causing them to disappear.

However, the signal represented by the samples shows up in the frequency spectrum as sidebands around those harmonics, and they are attenuated by the $\sin(x)/x$ curve as well. To recover the original flat spectrum, a filter with a response opposite that of the $\sin(x)/x$ curve must be inserted into the path.

Now that I've reviewed many of the fundamental issues of A/D and D/A converters, you're well equipped to discuss specific converter technologies next month. 📧

David Tweed has been developing hardware and real-time software for microprocessors for more than 22 years, starting with the 8008 in 1976. His system design experience includes computer design from supercomputers to workstations, microcomputers, DSPs, and digital telecommunications systems. David currently works at Aris Technologies developing digital audio watermarking. You may reach him at dtweed@acm.org.

MIDI

FROM THE BENCH

Jeff Bachiochi

Part 1: It Ain't Just for Music Anymore



Ever think of turning your PC into a

recording studio? This month, Jeff starts a project that lets you identify the state of outputs through the sound you create using the MIDI sequencer program.



My first job in the electronics field was with one of the pioneer manufacturers of analog synthesizers, Electronic Music Laboratories. EML began producing laboratory-grade oscillators, filters, modulators, and envelope generators in the late '60s.

Early analog synthesizers were programmed with patch cords connecting individual modules. The modules were controlled by turning knobs to raise or lower pitch, timbre, and loudness.

A natural progression led to using a more recognizable control device—the keyboard—to alter sounds. You may be familiar with the Moog synthesizer.

Next, the personal-computer market began spreading its wings. Radio Shack and Commodore released competing computers.

At this point, analog synthesizers and digital personal computers were about as far apart as possible. But, that gap narrowed until digital effectively replaced analog synthesizers. To this day, however, you find synthesizer owners who swear by older analog technologies, much like audiophiles who claim that LPs are superior to CDs.

Digital synthesizers meant a comfortable marriage between the growing complexity of sound synthesis and the ease of controlling that environment. With the abundant supply of various computer platforms and digital sound

generators, it wasn't long before performers were asking, "Why can't all this stuff work together?"

That's how standards are born from the bottom up. The MIDI (musical instrument digital interface) standard was first discussed in the '70s, yet it took almost 10 years to develop into a plan manufacturers could agree with.

MIDI PROTOCOL

The MIDI protocol was designed to standardize the control of sound synthesis without encroaching on how manufacturers designed synthesis circuitry.

Each manufacturer could recognize (or produce) control data complying with the standard. Pitch, note on/off, and pitch bending describe functions relating to sound modifiers without delving into the synthesis circuitry.

The physical interface necessary for interconnecting MIDI devices is defined in the standard and is easily (and cheaply) implemented. It uses five-pin DIN connectors.

Three pins are used on the MIDI out side. Pin 2 is the cable shield, pin 4 is +5 V through a 220-Ω resistor, and pin 5 is an open-collector output from a UART that can drive up to 50' of cable.

On the MIDI in side, pin 4 (power) connects to an optocoupler's anode, and pin 5 (serial data) connects to the cathode. The cable shield (pin 2) is left unconnected to the MIDI in system, preventing ground loop problems. The optocoupler's output drives the MIDI in's UART and a second five-pin DIN (the MIDI thru).

The optocoupler prevents direct electrical contact between systems, while the MIDI thru allows for the daisy-chaining of instruments.

MIDI transmissions are similar to standard asynchronous data transmissions of 8N1. The catch is the data rate, which is 31.25 kbps.

If this rate doesn't ring any bells, I'm not surprised. It's not a standard data rate but somewhere between the more familiar 19.2 kbps and 38.4 kbps.

Why this number? Remember the MIDI standard goes back to the infancy of personal computers. At that time, UART chips couldn't do the high data rates we're accustomed to today. This rate was the fastest they dared.

Status byte / 1st Data byte	2nd Data byte	3rd Data byte	
80...8F	Chan 1...16 Note off	Note number 0-127	Note velocity 0-127
90...9F	Chan 1...16 Note on	Note number 0-127	Note velocity 0-127
A0...AF	Chan 1...16 Polyphonic	Note on, Note number 0-127	Aftertouch 0-127
B0...BF	Chan 1...16 Control/Mode change	Function 0-127	Value 0-127
C0...CF	Chan 1...16 Program change	Program 0-127	None
D0...DF	Chan 1...16 Aftertouch	Value 0-127	None
E0...EF	Chan 1...16 Pitch wheel control	LSB 0-127	MSB 0-127
F0	System exclusive	Vendor ID	Data...Data
F1	Undefined	None	None
F2	Song position	LSB	MSB
F3	Song select	Song 0-127	None
F4	Undefined	None	None
F5	Undefined	None	None
F6	Tune request	None	None
F7	End of System exclusive (EOX)	None	None
F8	Timing clock	None	None
F9	Undefined	None	None
FA	Start	None	None
FB	Continue	None	None
FC	Stop	None	None
FD	Undefined	None	None
FE	Active sense	None	None
FF	System reset	None	None

Table 1—MIDI commands include status bytes (i.e., the first data bytes) and potential second and third data bytes. Adapted from D. Valenti, "MIDI by the Numbers," *Electronic Musician*, February, 1988.

MIDI FILE FORMAT

The sequence of events composing the mechanics of a performance—live with MIDI output or manually scored through a MIDI editor—can be saved to a file. These recorded instructions can recreate the performance whenever they're played into a MIDI instrument.

The MIDI file format is an important part of the standard. I'll just touch on some main points here so we can make use of it. For more information, examine the complete MIDI standard.

The MIDI file format consists of a header chunk followed by track chunks. A chunk is a packet of information.

The header chunk consists of the length (double word), the format (word), ntraks (word), and division (word). The double-word length shows the number of data bytes to follow. The format word is 0, 1, or 2, indicating a single track, simultaneous tracks, or sequentially independent tracks.

The number of ntraks tells how many chunks are in a file. The division

word describes how information is handled. If the division's most significant byte is positive, the division value is the delta-times equal to a quarter note. If it's negative, it indicates the number of frames per second. The least significant byte is the number of divisions in the frame.

Track chunks follow the header chunk and contain status (function)

Octave Number	Note Number											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Table 2—The MIDI specification only defines note number 60 as middle C, and all other notes are relative. The absolute octave number designations shown here are based on middle C = C4, which is an arbitrary assignment.

and data bytes. The data format depends on the status byte. Table 1 gives a partial list of status and data bytes.

Many of the last status bytes (F0-FF) are common real-time system messages. They require no data bytes and are normally followed by a new status byte.

Some meta-events also begin with FF. But because they contain additional data, they can't be confused with a system message since the character after FF won't have its most significant bit set.

When a header or track chunk includes a length, a special format called variable-length quantity is used. The value is represented by four bytes. All but the last significant data byte has its eighth bit set to a 1.

To determine the double word's actual value, drop all four of the most significant bits and right-justify the remaining 7-bit data into a 28-bit value (talk about overcomplexity!).

SOUND CARDS

Chances are, your PC isn't silent. PCs have evolved into multimedia devices, so most have sound cards installed.

That sound card came with a bunch of utilities. One of the first things you may have done was personalize your PC with sound (.WAV) files—perhaps as simple as a dong when a file is opened, or as elaborate as Captain Kirk asking Scotty for more power.

Sound cards let you play audio files and CD music from your machine's CD player. They also enable audio recording, sampling the audio at a high rate, and producing memory-intensive .WAV files.

Today, many sound-card manufacturers implement MIDI as well. A MIDI sequencer program is usually bundled with the drivers accompanying the card.

MIDI files are generally smaller than .WAV files. They contain only the notes to play and when to play them, not how they should sound (that's a job for the MIDI instrument).

The sequencer program turns your PC into a recording studio. It contains three basic parts—the score, the MIDI list, and the mixer.

With these tools, you can record a performance from a MIDI instrument via the MIDI interface, save it, edit it, and play it back through the sound card (used as a MIDI instrument) or through an external instrument via the MIDI interface.

Or perhaps you'd like to write music. The score tool displays the MIDI performance in black on white. That's black notes on a white page, including the musical terms necessary for scoring the complete performance.

Although a live performance is automatically scored, you can start writing music with a blank score. Just click on the notes and place them on the musical staff. But don't expect random notes to sound like real music. It takes many long hours to learn this skill.

I hope you're not reading this because you wish to be a composer. And

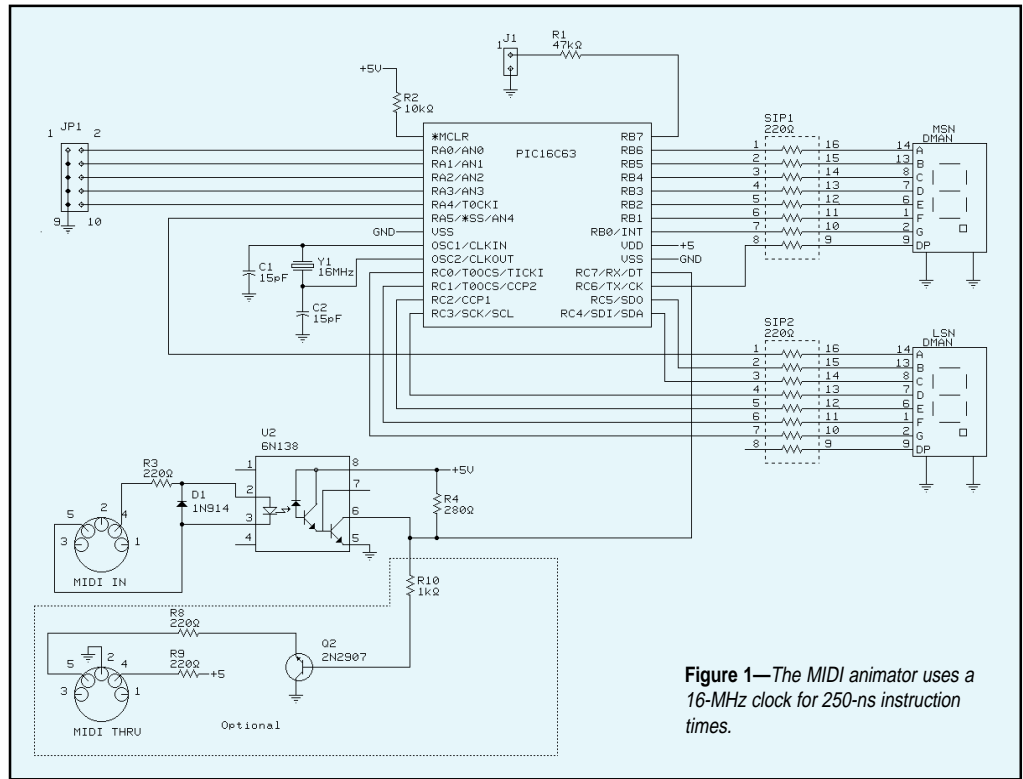


Figure 1—The MIDI animator uses a 16-MHz clock for 250-ns instruction times.

if you're still with me, you may wonder where I'm going with this. Well, this bit of background is necessary for my project. After all, what's a controller without a control language?

Your MIDI sequencer is the conductor in control of the MIDI animator. This project enables you to determine the state of outputs via the musical score you prepare using the MIDI sequencer. More on this later. First the hardware.

MIDI ANIMATOR

To receive MIDI commands, one must be able to receive a serial bit-stream of 31.25 kbps. Using a processor without a hardware UART requires a bit polling time of 32 μs.

By the time the sampling code is executed, there's hardly any time left between bits to accomplish anything, not to mention dealing with a packet size of up to 128 characters. I need a hardware UART to provide about 320 μs to do something useful in.

For this project, I chose a PIC16C63,

which has the required UART and plenty of I/O bits (see Figure 1). I use 15 I/Os as outputs (one for an LED) and seven as inputs (five for configuration, one for zero crossing, and the RX input for MIDI input).

All 14 control outputs will be PWM. I chose to use PWM so they can serve as both digital outputs (at 0% and 100%) and as pseudoanalog outputs for phase control of lighting.

Use the zero-crossing input to sync up with the AC line. If it isn't used, the software detects its absence during powerup, and the internal timer runs with no sync at a slightly slower rate.

Configuration jumper 4 chooses PWM or a modified RC servo mode. In RC servo mode, all outputs are 1–2-ms pulses, which supports 0–180° (or 0–90°) control of the common RC actuators.

Let's see how this works. The MIDI output I'm interested in contains note information. It comes in a track chunk.

Although any other MIDI commands will be tossed out, I still have to keep track of them to locate the ones I want. If a MIDI command's channel number matches the jumper selection I made on JP0–3, the command is recognized as legal for this

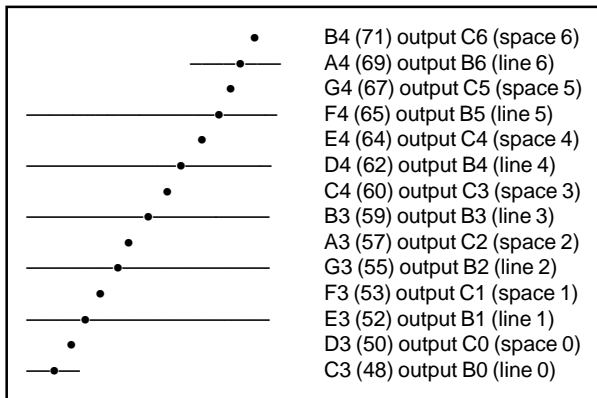


Figure 2—Here is the score for the MIDI animator. For simplicity, only the natural notes are used. The lines of the staff are used to output to one port, and the spaces are used to output to the other port.

Photo 1—The first step in this project is to display MIDI note commands. The seven-segment displays indicate legal hexadecimal commands broadcast over a MIDI interface.

circuit, enabling the user to control 16 MIDI animator circuits with just one score.

Once the channel number is OK'd, the note number identifies which output to act on. The musical score allows 0–127 notes.

I implemented C3 (octave three) through B4 (octave four), which covers seven spaces and seven lines of the standard treble-clef staff (see Figure 2). To keep the scoring simple, I didn't use sharps or flats. Table 2 lists available notes and their corresponding outputs.

Each note has an accompanying velocity value. Normally, this value is 64 (for instruments without velocity or aftertouch). I use this as the on (100%) indicator. Zero indicates off (0%). Numbers between 0 and 64 translate into a PWM output relative to the value.

MIDI MONITOR

It's always nice to have some small successes in a new design, but sometimes you need to alter your direction somewhat. Going from concept to finished form in one step can be more frustrating when it doesn't work. It's more satisfying to have intermediate goals whenever possible.

With 14 outputs available, I can connect two seven-segment displays and output any characters received as a two-digit hex value, one digit on each port. The code uses a look-up table to determine the state of the two 7-bit outputs for the upper and lower nibbles of the UART's receive buffer.

Since the characters come in faster than you can see them, a ring buffer holds the characters until they are called for by using the CFG4 jumper. The display routine samples the CFG4 input and waits for a low before displaying the next character.

That way, you can manually cycle through the buffer at your leisure without losing any data (assuming it doesn't overrun, which it probably will if you have a lot of activity). If the CFG4 input is held low, the display runs at full speed to keep up with the buffer.

One of the decimal points serves as a separate visual indicator tied to the TX output. It flickers whenever the RX interrupt routine is entered, indicating that things are working well.

The receive character routine checks to make sure the received character has no framing or overrun errors, and it places the received character into the ring buffer. At this point, we're not trying to interpret the data. We just want to grab and display it.

UNCONCLUSION

Next month, I'll discuss the PWM and servo-control feature and what devices you might want to add to the outputs. Here's your chance to start experimenting with your computer. This interface allows for all kinds of possibilities. Just think about it. ☺

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.

REFERENCES

www.midi.org
www.harmony-central.com/MIDI
nctnico.www.cistron.nl/midi.htm

SOURCE

PIC16C63
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 786-7277
www.microchip.com

SILICON UPDATE

Tom Cantrell

MegaMicro Card



In the world of data flash cards, Tom

compares MultiMedia Card to Compact-Flash and Nexcom. His result bodes well for the smaller, faster, and more versatile MMC in a variety of handheld products.



o place is Moore's Law more rigidly enforced than in the memory market. The name of the game: make it denser, smaller, faster, and cheaper—or go home.

Hope, at least from the supplier's perspective, can be found in the historic fact that ever-lower priced bits are quickly consumed. Program bloat gets much of the blame since it's easy to point the finger at some faceless programmer who couldn't resist filling another meg with frivolous features. But, exploding data is equally to blame, and nobody's forcing you to stockpile all those TIFs, GIFs, JPGs, and PDFs.

With the recent unveiling of the MultiMedia Card (MMC, shown in Photo 1) by SanDisk and Siemens, it's a good time to check out the flash data-card niche. But first, a bit of history.

Back in November '96, I described the battle between SanDisk Compact-Flash (CF) and the Intel-backed Miniature Card ("Flash Fight Flares," *INK* 76). Since then, it's safe to say that Compact-Flash is clearly winning.

In particular, widespread adoption of CF in the emerging digital camera market bodes well for the future. CF is designed into dozens of models, and SanDisk shipped more than a million CF cards in '97.

Until now, acceptance of these cameras has been held back by limited resolution. However, the emergence of mega-pixel models under \$1000, not to mention low-cost, photo-grade, inkjet printers are harbingers of big biz down the road.

Then, in August '97, I covered the serial flash module from a startup called Nexcom ("Serial Flash Busts Bit Barrier," *INK* 85). Checking up, I find that late last year the company, product line, and single-transistor memory technology were acquired by Integrated Silicon Solutions Inc. (ISSI).

These two articles set the stage for MMC, which can aptly be described as a hybrid that fills the gap between CF and the Nexcom-now-ISSI module. Simply put, MMC combines multi-megabyte aspirations with minimalist form factor, power consumption, and interface.



Photo 1—Memory marches on, and the MultiMediaCard, which crams many megabytes into its miniscule package, proves it.

Pin #	MMC Usage	SPI Usage	SPI Description
1	RSV	CS	Chip select (active low)
2	CMD	DataIn	Host-to-card commands and data
3	VSS1	VSS1	Supply voltage ground
4	VDD	VDD	Supply voltage
5	CLK	CLK	Clock
6	VSS2	VSS2	Supply voltage ground
7	DAT[0]	DataOut	Card-to-host data and status

Table 1—The MMC offers two interfaces (SPI and MMC) to enable a price/performance tradeoff that covers a broad range of applications.

BIG & SMALL

At only 32 mm×24 mm (and 1.4 mm thick), the MMC footprint is barely 1 in², which is about half the size of CF. In fact, the MMC occupies about the same area as the Nexcom module, although the aspect ratio is a bit more squarish.

However, where the Nexcom module topped out at a megabyte or so, MMC starts at 2 MB and goes all the way to 10 MB, with talk of 15- and 20-MB units in 1999 being bandied about. Meanwhile, CF is at 40 MB and headed to 80 MB. That is, MMC seems to be tracking at about a quarter of the capacity of CF.

Besides obvious data-storage applications like digital photos or voice recording, new ideas are emerging to take advantage of flash-card technology. For instance, a museum in Japan provides a hand-held audio player with a card storing the equivalent of a tour guide. A change in exhibits simply calls for updating the flash cards with the new info.

A subtle, but I suspect profound, difference for MMC is the prospect of read-only versions (i.e., ROM). Siemens has announced 2- and 8-MB units with plans for 32 MB in 1999 and a whopping 128 MB by 2001.

This announcement opens the door for MMC as a medium for distributing software of all kinds, including programs and reference data such as maps, phone books, and even music. Siemens uses the analogy that flash MMC is like a hard drive, while ROM MMC is like a CD-ROM.

The portable apps best served by the small size of the card are also likely to

be finicky about battery life. To that end, the MMC adopts a number of power-saving features.

To start, there's no 5-V option like the one offered with CF. Instead, the MMC operates at a somewhat lower voltage.

Actually, the card is required to be able to establish basic communication with the host over

a wide 2.0–3.6-V range. This communication enables the host to interrogate the card's Operating Condition Register (OCR), which defines the allowed voltage range (typically greater than 2.7 V) for memory access.

The amount of power consumed during memory access isn't trivial (e.g., 35 mA at 3.3 V). However, the MMC has a low-power standby mode that cuts power by a factor of almost 1000 (e.g., 50 µA at 3.3 V).

The host can overtly issue a command that causes the card to go into standby. However, it may not be necessary because the MMC automatically puts itself to sleep after 5 ms of inactivity.

There's no need to reset the card or otherwise go through hoops to get going again. Even in standby mode, the card remains conscious enough to detect a subsequent command and wake itself up. Only a 1-ms delay is required before the card is ready for the next read or write, as opposed to the 50-ms delay that is required after powerup.

MMC	SPI
Three-wire serial data bus (clock, command, and data)	Three-wire serial data bus (clock, data in, and data out) and card-specific CS signal
Variable clock rate 0–20 MHz	Variable clock rate 0–5 MHz
Up to 64k cards addressable by the bus protocol	Card selection via a hardware CS signal
Up to 30 cards stackable on a single physical bus	Up to 10 cards stackable on a single physical bus
Easy card identification	Not available
Error-protected data transfer is available	Optional. A nonprotected data-transfer mode.
Sequential and single/multiple block-oriented data transfer	Single block read/write

Table 2—Using the simpler SPI mode sacrifices some functionality related to addressing and multiblock data transfers. But, the difference between the modes is an issue for designers, not users, because the card uses whichever interface is requested by the host it is currently plugged into.

TWINTERFACE

One factor that really differentiates the MMC from CF is the interface. CF, reflecting its PCMCIA roots, requires a whopping 50 pins to support its 8-/16-bit IDE-disk-drive-compatible bus. Needless to say, the size and cost goals of MMC demand something more streamlined, as in just seven signals.

As opposed to the expensive and mechanically precise pin-and-socket-style connector of PCMCIA and CF, the MMC uses the surface-contact slide-in approach like the Nexcom module.

A close look at Photo 1 shows that the socket power and ground contacts are offset to connect first on insertion and disconnect last on removal—a basic requirement for hot swap.

Taking power and dual grounds may leave only four pins to get the job done, but SanDisk manages to provide two rather different ways of doing so (see Table 1).

I can imagine how the arguments went. On one side, the purists arguing for an elegant new interface offering high speed and lots of neat capabilities. On the other, pragmatists willing to dispense with the bells and whistles in favor of something quick and easy to hook up to any micro. Tastes great or less filling? Why not do both?

The purists get what they want with the so-called MMC interface, which is the default when the card powers up. Pragmatists can choose a simple SPI (i.e., clocked serial) interface that, at best, directly connects to the ever-growing list of so-equipped micros or, at worst, calls for a few lines of bit-banging code.

How does the MMC know which interface to use? In MMC mode, pin 1 is a reserved No Connect, but it's defined as Chip Select (CS) for SPI mode. At powerup, the MMC card checks the CS pin and, if it's asserted, switches the interface from MMC to SPI mode.

In both modes, pin 5 is the clock input generated by the host to time data transfers. Data is referenced to the falling edge of the clock.

The difference between the modes largely boils down to

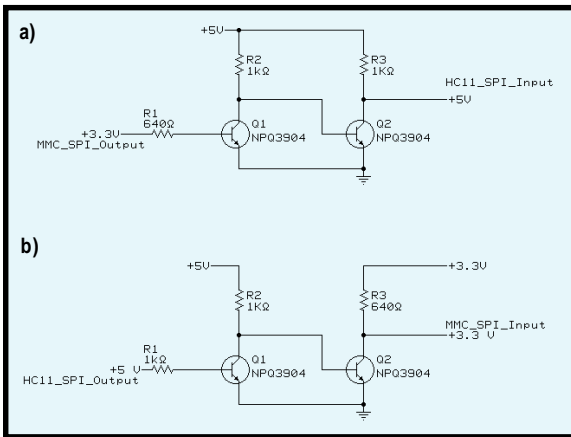


Figure 1—The fact that MMC is low voltage only (2–3.6 V) may dictate the use of level shifters. One approach that works with a 5-V SPI port uses transistor pairs to (a) step up MMC outputs and (b) step down MMC inputs.

toward lower operating voltages to reduce power consumption and extend battery life. In situations where the MMC must connect to a 5-V device, level shifters are required. Figure 1, taken from a SanDisk app note, shows transistor pairs configured to step up and step down a 5-V SPI interface.

SMARTS ONBOARD

SanDisk sticks with the strategy of using an onboard intelligent controller in front of the memory chip. Their success in the marketplace is the best argument for adding a controller which, despite the cost penalty, is more than offset by various benefits.

The controller goes out of its way to help preserve data integrity, incorporating functions like internal ECC, CRC, bad sector mapping, and wear-dependent write algorithms (write endurance is 300k cycles). Because the card handles these important functions, you don't have to fuss with them.

the last two pins. For MMC mode, they function as bidirectional command (CMD) and data (DAT) lines, while for SPI they are unidirectional data lines (DIN and DOUT). Also, the MMC CMD line switches between open-collector and push-pull output configuration, while SPI is push-pull only. Table 2 sums up the functional differences between the modes.

While SPI requires a CS line for each card, MMC mode uses an addressing scheme that supports, logically at least, up to 64K cards in a stack. Here's how.

Each card has a unique 128-bit card ID (CID) register. In response to an ALL_SEND_CID broadcast from the host, all attached MMC cards try to drive their own CID on the CMD line (open-collector mode), and each simultaneously monitors the line for comparison. Any time a card outputs a 1 but sees a 0, it backs off (i.e., quits sending its CID).

By the time the host clocks in the last bit of CID, only one card is left standing. The host proceeds to assign that card a 16-bit relative card address (RCA) that is used for the duration of the session.

The host keeps issuing ALL_SEND_CID commands. Cards that have gotten their RCA remain quiet. Eventually, all cards have an RCA and the last issued command times out, signaling completion of the ID phase.

Another major difference is that MMC mode, thanks to the ability to overlap commands and data, offers terminate-at-will multiblock and streaming transfer modes. SPI handles everything as a single-block transfer with predetermined length.

Finally, the MMC mode offers faster raw transfer (20 MHz vs. 5 MHz for SPI). However, the advantage of the higher speed is mainly found with the multiblock and streaming modes. The actual throughput is ultimately limited by memory bandwidth: 1 MBps for reads and 200 kbps for writes.

The fact that the MMC doesn't have a 5-V option is evidence of the trend

SPI Index			
Cmd	Mode?	Abbreviation	Description
CMD0	Y	GO_IDLE_STATE	Resets all cards to idle state
CMD1	Y	SEND_OP_COND	Request and confirm operating conditions
CMD2	N	ALL_SEND_CID	Request all cards send their ID number
CMD3	N	SET_RELATIVE_ADDR	Assign 16-bit relative card address (RCA)
CMD4	N	SET_DSR	Select output driver configuration
CMD7	N	SELECT/DESELECT_CARD	Select addressed (RCA) card
CMD9	Y	SEND_CSD	Request addressed card to send CSD data
CMD10	Y	SEND_CID	Request addressed card send ID number
CMD11	N	READ_DAT_UNTIL_STOP	Stream read
CMD12	N	STOP_TRANSMISSION	Stop stream read
CMD13	Y	SEND_STATUS	Request addressed card send its status
CMD15	N	GO_INACTIVE_STATE	Set addressed card to inactive
CMD16	Y	SET_BLOCKLEN	Set block length for block commands
CMD17	Y	READ_SINGLE_BLOCK	Read a single block
CMD18	N	READ_MULTIPLE_BLOCK	Read multiple blocks
CMD20	N	WRITE_DAT_UNTIL_STOP	Stream write
CMD24	Y	WRITE_BLOCK	Write a single block
CMD25	N	WRITE_MULTIPLE_BLOCK	Write multiple blocks
CMD26	N	PROGRAM_CID	Program card ID (factory use only)
CMD27	Y	PROGRAM_CSD	Protection writable bits of CSD register
CMD28	Y	SET_WRITE_PROT	Protection on for addressed group
CMD29	Y	CLR_WRITE_PROT	Protection off for addressed group
CMD30	Y	SEND_WRITE_PROT	Request card protection status
CMD32	Y	TAG_SECTOR_START	First sector in erase list
CMD33	Y	TAG_SECTOR_END	Last sector in erase list
CMD34	Y	UNTAG_SECTOR	Remove sector from erase list
CMD35	Y	TAG_ERASE_GROUP_START	First group in erase list
CMD36	Y	TAG_ERASE_GROUP_END	Last group in erase list
CMD37	Y	UNTAG_ERASE_GROUP	Remove group from erase list
CMD38	Y	ERASE	Erase all previously selected sectors
CMD39	Y	FAST_IO	Access app-specific (non-MMC) registers
CMD40	N	GO_IRQ_STATE	Enter interrupt mode
CMD59	Y	CRC_ON_OFF	Enable/Disable CRC (SPI mode only)

Table 3—The MMC command set offers dozens of high-level commands dealing initialization and configuration, data transfer (block, multiblock, and stream), and erasure and write protection.

Perhaps the most important benefit of the separate-controller approach is that it decouples the host-system hardware and software design from the particulars of the underlying memory technology.

The host issues high-level commands like read, write, erase, and so forth, and the controller handles the details. This means today's design will work with tomorrow's MMC cards, no matter what kind of esoteric memory technology finds its way under the hood.

Along with the IDE-disk drive pretensions, the rigid adherence to disk nomenclature (cylinder, head, etc.) that characterized the earlier card's interface is fading. Other than the fact that the basic building blocks are 512-byte sectors, the organization isn't really much like a disk at all.

Instead, the memory is partitioned as shown in Figure 2. The smallest amount that can be erased defines a sector. An erase group comprises 16 sequential sectors.

A single erase command can zap an arbitrary selection of the sectors (i.e., any or all of the 16) within an erase group or an arbitrary selection of erase groups. The process involves tagging the start and end of a sequence of sectors or erase groups, untagging those (up to 16) that aren't to be erased, and then issuing the erase command.

A write-protect group, the smallest individually protectable unit, is composed of 32 erase groups. Plus, two card-level write-protect bits—one temporary and one permanent—offer global protection.

There's a permanent copy bit that is presumably intended to combat piracy. However, protection seems to depend on trusted software on the host or programmer, because the copy-bit setting doesn't otherwise affect card operation.

The global write- and copy-protection bits are found in the CSD (card-specific data) register which, like the previously mentioned CID and OCR (voltage profile) registers, defines various card-unique parameters such as speed, power requirements, and partitioning (i.e., block, sector, group, device sizes).

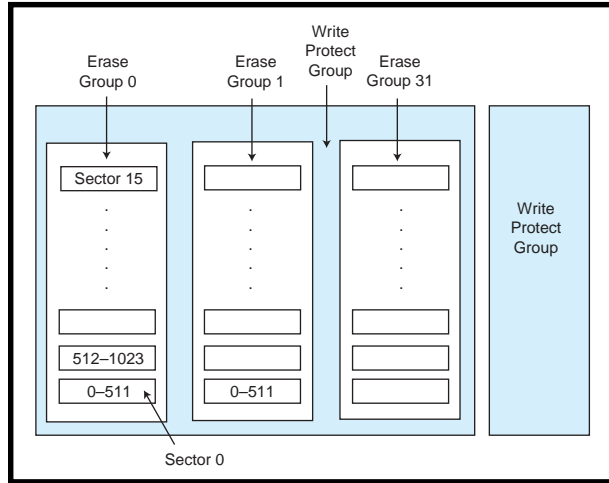


Figure 2—The basic unit of storage on an MMC card is called a sector, but that's about the extent of any resemblance to a disk drive.

Table 3 summarizes the commands that are handled by the MMC. The subset of commands related to multi-block transfer and software addressing aren't available in SPI mode, as I described earlier. All of the commands are six bytes in length, while responses vary from 1 to 16 bytes, depending on the command as well as on which interface is being used.

STACK THE DECK

If you think the MMC is an ace, a good place to see more is the recently formed Multi-Media Card Association. You'll find late-breaking info and links to members like SanDisk, Siemens, Hitachi, Motorola, Nokia, and others.

A \$340 evaluation kit from SanDisk comes with a 20-MB MMC card, PC parallel port MMC drive, extender card, and the requisite software utilities and documentation.

The MMC is less PC-centric than earlier cards, and it targets many nonPC-related apps.

But, there's no denying the PC is often at least one, if not the ultimate, destination for just about all data.

Unlike earlier disk cards, MMC can't piggyback on the PC's built-in IDE support. To make MMC look like a disk, you need flash-file-system software. SanDisk offers a \$2545 host developers toolkit containing the C source for a FAT (file allocation table) file system.

pc_cluster_size	Return cluster size	pc_rmdir	Delete a directory
pc_diskabort	Abort operation	pc_set_attributes	Set file attributes
pc_dskclose	Flush FAT and files and free buffers	pc_set_cwd	Set current working directory
pc_diskflush	Flush FAT and files	pc_set_default_drive	Set default drive specifier
pc_format	Format card	pc_stat	Get file or directory statistics
pc_free	Return bytes free on card	pc_unlink	Delete a file
pc_fstat	Return statistics on open file	po_close	Close a file
pc_gdone	Free pc_gfirst and pc_gnext resources	po_extend_file	Extend a file
pc_get_attributes	Get file attributes	po_flush	Write a file directory entry and flush FAT
pc_gfirst	Return first entry in a directory	po_lseek	Move the file pointer
pc_gnext	Return next entry in a directory	po_open	Open a file
pc_isdir	Test if path is a directory	po_read	Read from a file
pc_mv	Rename a file or directory	po_truncate	Truncate a file
pc_pwd	Return the current working directory	po_write	Write to a file

Table 4—The SanDisk host-developers toolkit software provides a FAT file system-compatible Application Programmers Interface (API).

Porting the driver to a design starts with writing a minimal set of low-level hardware-specific drivers that establish physical communication with the MMC. A configuration file specifies a variety of options, such as buffer sizes, whether to preerase when a file is deleted or extended (speeds subsequent writes), MMC or SPI interface, and so on.

Put it all together, and you end up with an API (see Table 4) that knows about disks, files and their attributes, directories, and so forth.

GIGANOCARD?

Compared to older cards, MMC is a better fit with the form-factor, power-consumption, and cost requirements of anything that purports to be handheld or fit in a pocket.

I especially like the simple, versatile interface. It's nicer to deal with a few pins rather than the 50+ of earlier cards. Thanks to SPI mode, the card is easily managed by the lowliest processors, yet designs requiring performance can exploit the faster MMC mode.

SanDisk offers 2-, 4-, and 8-MB cards at \$26, \$32, and \$43 in volume (for now, it looks like the formula is \$3 per MB + \$20). As of today, capacity, price, or both may hold back some applications. But thanks to Moore's Law, both concerns will diminish over time, broadening MMC acceptance and design-in.

The interesting question is, what happens next? History would predict that another downsizing lies around the corner. The only problem is that while silicon may shrink, people won't.

The wizards may get a zillion bits on the head of a pin, but that doesn't mean it's wise. Make the thing much tinier, and you'll need a magnifying glass and tweezers to boot up. ☹

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.

SOURCES

MultiMedia Card

SanDisk Corp.
(408) 542-0500
Fax: (408) 542-0503
www.sandisk.com

Serial flash module

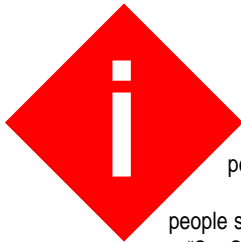
Integrated Silicon Solutions, Inc.
(408) 588-0800
(408) 588-0805
www.issiusa.com

MultiMedia Card Assn.

(408) 253-0441
Fax: (408) 253-8811
www.mmca.org

PRIORITY INTERRUPT

Banking on Bugs



really have to be more careful about destroying my image. After all, if perception is 99% of reality, why mess with people's perception?

It all started at a party. It was your basic, eat, drink, and be merry, business-acquaintance get-together where people split off into little groups to discuss subjects that typically require a two-cocktail prologue.

"So, Steve, I'm told you're a magazine publisher?" I'm not sure what smelled stronger, the smoke from the London broil on the grill or his martini breath. The problem with all these "get to know ya" business parties is that inevitably you are asked a question about professional rank by a person who can't conceivably understand the answer. When you've been self-employed as long as I have, you can pretty much call yourself anything you want—President, Engineer, Salesman, Publisher, Editorial Director, even Janitor. When an investment banker has had a couple martinis and asks if you're a magazine publisher, you definitely have to be careful.

Just like there are people who think food comes from grocery stores, there are professional people who use computers every day without ever considering how they're designed or manufactured. Experience has taught me that these people associate the word publishing with McGraw-Hill, Time, and Rupert Murdoch. Ultimately, it's counterproductive to shatter their lofty image with cold reality.

I'd love to say (even just once), "I'm the janitor," but usually I cop out and simply say, "I'm involved a bit in publishing, but I really prefer to think of myself as a design engineer." Thankfully, the information age has educated bankers so that I no longer have to add, "and I don't build bridges." Of course, now they think we're all computer engineers (whatever that is), and to them, "computer" only means PCs!

The conversation went back and forth a few times as I tried to explain about embedded control (definitely a mistake). He admitted that PCs certainly weren't in everything, but he just couldn't grasp the concept of single-chip computers in things like toasters and power tools. At this point, mere explanation was becoming a challenge. I passed my basting brush to the person closest to the grill and said to the banker, "Obviously, the only way is to show you. I have a microcontroller design over in the shop. Come on."

A half dozen people ended up trekking over to the workshop. As we descended the stairs a couple of them hesitated. I chose not to tell them why this project was located here and not in the Circuit Cellar. I didn't want to confuse the issue. Soon it would become clear to them.

The mixed clutter of electronic equipment, power tools, and carpentry devices presented an air of eclectic insanity. I could sense they were reconsidering their descent into the dungeon.

"It's OK, just step over that stuff. And, watch out for those wires! They're probably live!" (They weren't, but there are times when it's just fun to say that, especially to bankers).

We walked around a workbench and stopped in front of an equipment cart piled high with electronics. An assortment of pulse generators, oscillators, and amplifiers were intertwined to produce a complex signal, which appeared as a rapidly changing sweep frequency on the brightly lit oscilloscope. (I didn't even try to explain sweep frequencies or oscilloscopes to them.) They seemed hypnotized by the pulsating hum of the electronics combined with the strobe-like rhythm of the oscilloscope. That was, until one of the ladies yelled, "It's full of insects!"

Immediately, they jumped back. The banker looked at the 7' plexiglass enclosure that was indeed full of six-legged critters. His startled expression said it all. Embedded control designers must be real fruitcakes.

"No! You don't understand. Yes, the case is full of insects! In fact, we used a bunch of rodents before that..." I could sense the hole getting deeper.... "That's what we're designing! Wait, that's not what I mean!"

I moved quickly to block their exit and explain, "A while ago we designed a commercial device that repels rodents. Inside it, there's a microcontroller." I held out a tiny chip in my hand. At least now I had their attention.

"All this equipment simulates the signal that we squashed down into this chip. [Of course, you all know better but sometimes you have to lay it on thick for bankers.] Testimonials from customers said that it did work on rodents, but it also seemed to drive out the insects. We decided to test it." There was a silent pause as everyone gazed at the festering bugs.

"Don't worry. They can't escape [I hope]." Suddenly, my choice of location was clear to them. *Who wants all these bugs in the house?*

Having assured them of the enclosure's security, they began to relax a bit. They even conceded that testing a product was a commercial necessity. But as bankers and financial people, they just didn't seem to grasp the significance of a dedicated microcontroller or the value in it. That was until the investment banker added, "So, does anybody buy this thing?"

I looked at him and grinned, "How does 50,000 a month strike you!" It was like a universal language translator had just been introduced to the communication. Embedded microcontrol was instantly understood as high volume and big bucks. Added explanation was unnecessary.

As we walked back to the party, the banker seemed a little more animated. Obviously, my not being a publishing tycoon was acceptable. He smiled as he elbowed a little closer and whispered, "So, Steve, you need any money?"