www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#119 JUNE 2000

ANALOG TECHNIQUES

Analog Amplifier Design Tips

Quartz Crystals and Oscillators

Digging Into Op-Amp Specifications

A Motorized Telescope Mount

CIRCUIT CELLAR ONLINE

Double your technical pleasure each month. After you read *Circuit Cellar* magazine, get a second shot of engineering adrenaline with *Circuit Cellar Online*, hosted by ChipCenter.

— FEATURES —

Monitoring Your Micro
Daniel Mann and Jim Magro
If you're going to keep your system running at its peak performance, it helps to be able to monitor the microprocessor. Most processors use counters to measure performance, but Daniel and Jim have a better way of gathering and analyzing performance data.

The Ultimate 16-bit Microcontroller
Robbert Maris
Memory organization, often ignored by chip vendors, significantly affects code density and speed. This is especially important in applications that involve user interface or extensive communications. Robbert outlines a solution that may reduce controller diversity in companies, hence reducing your tool investment.

MPEG and DSP Integration
Priyesh Surati and David Austin
For their final project at the University of Calgary, Priyesh, David, and their team members wanted to demonstrate MPEG decoding on Analog Devices' SHARC 21061. Here, they share what they learned about MPEG history, MP3, and the music industry's future.

— COLUMNS —

Learning the Ropes
Charming Adders
Implementing an Adder in an FPGA
Ingo Cyliax
Designing with FPGAs is similar to working with other digital components, and opens the door to interesting design permutations. This month, Ingo presents how structures are implemented in FPGA logic blocks.

Lessons from the Trenches
The Race Is On
Catching Internet Connectivity Fever
George Martin
Do you want your toaster hooked up to the Internet? It may sound absurd, but people are racing to connect their households. Until now, even thinking about it was daunting, however, George discovered an easy way to get "iConnected".

Silicon Update Online
It's Two, Two, Two Memories in One
Tom Cantrell
The quest for the best memory is endless. You know about Silicon Storage Technology's combination chips SST38 and SST30. Tom takes a closer look at SST's MCP marvel that combines Flash and SRAM on one chip.

Resource Links
- Wireless Application Protocol (WAP)
- Making of PDF and Postscript Files

Bob Paddock

Test Your EQ
8 Additional Questions

WWW.CIRCUITCELLAR.COM/ONLINE
Table of Contents for May 2000

Coming Soon....
Circuit Cellar Online 1999 issues will be available on CD. The CD will contain all the online files, the PDF files and any referenced code files for issues July 1999 through December 1999.

Also on the CD are the Embedded Internet Workshop files from years 1998 and 1999.

THE ENGINEERS TECH-HELP RESOURCE

Let us help keep your project on track or simplify your design decision. Put your tough technical questions to the ASK US team.

The ASK US research staff of engineers has been assembled to share expertise with others. The forum is a place where engineers can congregate to get some tough questions answered, or just browse through the archived Q&A's to broaden their own intelligence base.

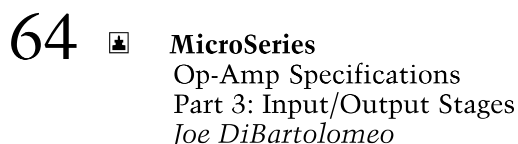
12 **Get It Right the First Time**
Tips and Tricks for Designing with Single-Supply Analog Amplifiers
Bonnie Baker

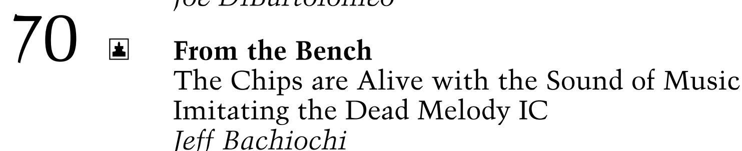
18 **Keep Up with the Stars**
Motorized Telescope Control
Mel Bartels

26 **Back to the BasicX**
Part 1: NetMedia's Development System
Brian Millier

34 **Right on Time**
Quartz Crystals and Oscillators
George Novacek

54 **Implementing a RAS Server Port**
Shawn Arnold


64 **MicroSeries**
Op-Amp Specifications
Part 3: Input/Output Stages
Joe DiBartolomeo


70 **From the Bench**
The Chips are Alive with the Sound of Music
Imitating the Dead Melody IC
Jeff Bachiochi

76 **Silicon Update**
On the Road Again
Part 2: Taking Silicon for a Test Drive
Tom Cantrell

INSIDE ISSUE 119

Task Manager 6
Rob Walker
What's it Worth?

New Product News 8
edited by Harv Weiner

Reader I/O 11

Test Your EQ 84

Advertiser's Index 95
July Preview

Priority Interrupt 96
Steve Ciarcia
Choice Versus Default

EMBEDDED PC

40 **Nouveau PC**
edited by Harv Weiner

42 RPC **Real-Time PCs**
Real-Time Executive for Multiprocessor Systems
Part 2: Running i386 RTEMS Applications
Ingo Cyliax

48 APC **Applied PCs**
Picking Some ExacTicks
Keeping Precise Time
Fred Eady

What's It Worth?

veryone knows that there's a difference between value and cost. I have an antique bookcase that was given to me by my grandmother. If you wanted to buy a similar bookcase it would cost you around $900. However, your $900 wouldn't move my bookcase an inch closer to the door.

Not too long ago, I was opening my mail and found a "Congratulations! You are one of the 100 lucky winners…" letter. I stopped reading and was all set to recycle it when a $10 Amazon.com gift certificate fell out. Suddenly there was some value to this letter.

I finished reading the letter and found out that I was one of 100 selected people who responded to a survey about how businesses use the Internet. To be honest, I didn't take the time to fill out the survey because I was interested in finding out how businesses use the Internet. I took the time to fill it out because: (a) I use the Internet (b) I often get frustrated with how businesses present themselves on the Internet, and (c) it was short enough to finish in about five minutes.

Obviously, the most important reason is (c), because time is rather valuable to me. However, (b) ranks right up there because time is rather valuable to me. Get on the Internet and you'll see that there are definitely some businesses out there that have no idea how customers or other businesses are using the Internet. So, if investing a few minutes of my time to fill out a survey can make my next trip on the Internet more profitable, then it was worth it.

Of course, that may be excessively optimistic. But, I think it's more reasonable than the idea of lobbying the government to find a way to provide everyone with access to computers and Internet technology. Attempting to minimize the digital divide is a great idea, but I'm not sure it's a process that should involve the government. Anyone who had to fill out the long census form will be glad to tell you that the government is already overly involved with personal and domestic issues.

You'll notice on page 11 of this month's issue that we revived the Reader I/O page. This section used to be a monthly feature, but for some reason, over the last few months, we've received a low amount of reader feedback, which is uncharacteristic for *Circuit Cellar* readers. Hopefully no news is good news, but because it has been awhile since we last had a Reader I/O page, I thought I'd remind everyone that you're always welcome to send in feedback (positive or negative). It may cost you a few minutes of time, but your comments are something we value.

Even if my survey response doesn't dramatically change the way businesses view the Internet, my next stop at Amazon.com will definitely be more profitable, thanks to a communications consulting firm that appreciates the fact that time is valuable.

P.S. The deadline for the Design2K contest is June 30, so if you've got an 8051 project in mind, it's not too late to enter. Check out the prizes and you'll see that it could be well worth your time!

Rob

rob.walker@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

MANAGING EDITOR
Rob Walker

TECHNICAL EDITORS
Jennifer Belmonte
Rachel Hill
Jennifer Huber

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Mike Baptiste Ingo Cyliax
Fred Eady George Martin
Bob Perrin

NEW PRODUCTS EDITOR
Harv Weiner

PROJECT EDITORS
Steve Bedford
Janice Hughes
James Soussounis
David Tweed

ASSOCIATE PUBLISHER
Sue Skolnick

CIRCULATION MANAGER
Rose Mansella

CHIEF FINANCIAL OFFICER
Jeannette Ciarcia

CUSTOMER SERVICE
Elaine Johnston

ART DIRECTOR
KC Zienka

GRAPHIC DESIGNER
Mary Turek

STAFF ENGINEERS
Jeff Bachiochi John Gorsky

QUIZ MASTER
David Tweed

EDITORIAL ADVISORY BOARD
Ingo Cyliax
Norman Jackson
David Prutchi

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

ADVERTISING

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

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

ADVERTISING CLERK Sally Collins

CONTACTING CIRCUIT CELLAR

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, 4 Park St., Vernon, CT 06066
AUTHOR CONTACT:
E-MAIL: Author addresses (when available) included at the end of each article.

For information on authorized reprints of articles,
contact Jeannette Ciarcia (860) 875-2199 or e-mail jciarcia@circuitcellar.com.

CIRCUIT CELLAR®, THE MAGAZINE FOR COMPUTER APPLICATIONS (ISSN 1528-0608) and Circuit Cellar Online are 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 Subscriptions, P.O. Box 5650, Hanover, NH 03755-5650 or call (800) 269-6301.

Postmaster: Send address changes to Circuit Cellar, Circulation Dept., P.O. Box 5650, Hanover, NH 03755-5650.

Circuit Cellar® 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® disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published by Circuit Cellar®.

The information provided by Circuit Cellar® is for educational purposes. Circuit Cellar® makes no claims or warrants that readers have a right to build things based upon these ideas under patent or other relevant intellectual property law in their jurisdiction, or that readers have a right to construct or operate any of the devices described herein under the relevant patent or other intellectual property law of the reader's jurisdiction. The reader assumes any risk of infringement liability for constructing or operating such devices.

Entire contents copyright © 2000 by Circuit Cellar Incorporated. All rights reserved. Circuit Cellar and Circuit Cellar INK are registered trademarks of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

NEW PRODUCT NEWS

Edited by Harv Weiner

MODULAR MOTOR CONTROLLER

DigiDrive II is a family of modular controllers that controls the speed of 1/10- to 1-horsepower single-phase induction motors. The family consists of a modular set of drive electronics that connects to supply single-phase motor drives for applications such as fans, pumps, and compressors. The controllers provide energy savings, performance improvement, increased reliability, along with extending motor life.

The controller consists of three subsystems. The power drive board (PDB) contains the power electronics, drive circuitry, and basic drive controls. The power interface (PINT) connects the motor and AC line to the power drive board (PDB). It contains filter and energy storage elements that meet agency approval standards for noise and safety. The control interface (COIN) provides specified control settings, interface communication ports, and flexibility in control settings and the interconnect system. Combining modules creates a family of controllers that meets the needs of single-phase systems.

DigiDrive uses analog, digital, power electronics, and proprietary microcontroller and software algorithms. Integrating provides better energy efficiency and performance. Firmware modifications and hardware selections result in faster time-to-market and lower overal cost.

Five-part samples are available for **$85**. Pricing for 1000-piece quanities starts at **$60**.

Anacon Systems, Inc.
(888) 456-3398
(512) 263-8668
Fax: (512) 263-8060
www.anaconsystems.com

NEW PRODUCT NEWS

EMBEDDED LINUX DEVELOPMENT PLATFORM

Linux Planet is an embedded Linux development platform that integrates RPX hardware and firmware with Hard Hat Linux from MontaVista. It shortens the time required to build embedded devices and is easy to use with standard hardware and software.

Linux Planet comes in a rugged, colorful enclosure with a 12-bit VGA resolution LCD with a touch-screen on top. The computing power is composed of the RPX Lite CPU engine and the H I/O X expansion module. The RPX Lite features the Motorola PowerPC MPC823 microprocessor, a 10-Mbps Ethernet port, PCMCIA, RS-232 interface, and an onboard ambient temperature sensor. The I/O card offers an MP3-performance capable CODEC, video out for NTSC/S-video, multiple serial ports, IrDA, and connections for the touchscreen. The RPX bus, composed of two standard 120-pin connectors that provide access to the PowerPC core and the communications processor module, connects the two. The RPX bus accommodates modules of different mechanical configurations and MPC8XX processors.

Hard Hat Linux includes cross and native development toolkits for PowerPC, *x*86/Pentium, and other processors. A start guide, Linux kernel configuration, scaling tools, device drivers, and sample code for the RPX are included. Also included are unlimited e-mail support for the hardware, a support program for the whole solution, cables, and a universal power supply.

Linux Planet costs **$5995** with a six-month subscription to Hard Hat Linux, and $7495 with a 12-month subscription.

Embedded Planet
(440) 646-0077
Fax: (440) 461-4329
www.embeddedplanet.com

READER I/O

THE OLD SCHOOL

I missed Steve by a few hours at the Embedded Systems Conference in Chicago, but I was pleased to see *Circuit Cellar* there. I think the keynote address from Clifford Stoll really spoke to the kind of material Steve's old articles in *BYTE* and current *Circuit Cellar* articles address. There are some of us (some days I think fewer and fewer) who really want to know how things work.

As a mechanical engineer who has worked with computers and software for his whole career, I find that I benefit from understanding or at least having some understanding of how the computer works from an EE's point of view.

Stoll recommended that we remove computers from classrooms and replace them with a box of parts from which we teach the kids to build computers. What a curious idea.

Indeed, the real-time assembler programming course I took as a student on a PDP-11-03 is still useful to me whereas the usefulness of my vast understanding of Wordstar evaporated in about two years. And, although the BASIC language I learned on a TRS-80 Model 1 still serves me today, CP/M commands don't seem to come in as handy. The more fundamental information, the longer the knowledge seems to last. What use is it to teach kids how to use Windows 95?

So keep up the great work! Maybe you could throw in some industrial style servo motion control material for those of us who dream of building an industrial CNC lathe in their basements.

Ian Jefferson

I enjoyed Steve's editorial in the March (116) issue. In our society's zeal to ensure that life is risk-free, we have sucked every bit of "chance" out of education and learning. If I hear, "If it saves one child's life or prevents one accident, it's worth it..." one more time, I'll shout.

Somehow the adventuresome, risk-taking American public decided or became convinced that it wanted the sensation of challenge and exploration, but without any risk (Mr. Columbus, can you absolutely guarantee that you will come back will all crew members safe and sound?). I think it's part of the Disney-fication of life—everything's planned, you feel a thrill, but there are no surprises.

I fear for the long-term consequences to our knowledge base and society. But sadly, I blame engineers as much as anyone else. Because we now have (and are proud of) the apparent know-how and tools to analyze and simulate in great detail and with high confidence (we think we can know everything before doing anything), we feel we must do so—and that the answers will be correct.

To make it worse, there are the huge numbers of oversight committees, groups, activists, post-event analyzers and Monday-morning quarterbacks, sensationalism-based news outlets, and so on microanalyzing every move in advance and afterwards, too. No moonshot attempt for these folks—someone might get hurt.

Thanks for letting me spout. I grew up in an apartment in New York City, had all sorts of chemicals in my room, plus a workshop with small power tools there—and also loved testing my .049 glow-plug airplane engines in my room. The noise, the smell, the risk—it was great!

I used to go to local "chemical" shops and buy whatever I needed, such as copper sulfate for copper-plating experiments. Try doing that now and you'll get arrested just for asking if they sell it.

Bill Schweber

Editor's Note: In the article "Accurate Linear Measurement Using LVDTs" in the May 1999 issue (106), there was a reference that implied that Schaevitz Technologies is associated with Macro Sensors. There is no relation between Macro Sensors and Shaevitz Sensors. The Handbook of Measurement and Control *is the property of Shaevitz Sensors, Hampton, VA.*

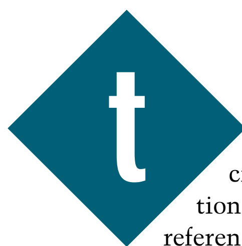
Get it Right the First Time

FEATURE ARTICLE

Bonnie Baker

Tips and Tricks for Designing with Single-Supply Analog Amplifiers

You can learn from your own experience (mistakes), or you can learn from someone else's. Bonnie makes the choice easy by sharing some of the insight she's gained in her years of analog design. Pay attention now or pay the price later.



The op-amp circuit descriptions found in most reference books present a computational algorithm that, theoretically, will provide the solutions to your analog amplifier design woes. With a perfect amplifier, these designs would be easy to implement. But there isn't a perfect amplifier, yet.

Throughout the history of analog system design, circuits have required special care in key areas in order to ensure success. Common sense and bench sense will pull you out of most potential amplifier design disasters.

In an ideal world, the perfect amplifier would look like the one described in Figure 1. With this perfect amplifier, the input stage would be designed with devices whose inputs (IN+ and IN–) can be taken all the way to the power supply rails. As a matter of fact, it would be nice if they operated beyond the rails.

And, the inputs would not source or sink current (i.e., they would have zero input bias current). Because voltage errors across the two inputs are usually gained by closed-loop circuit configurations around the amplifier, any DC voltage error (offset voltage) or AC error (noise) would be zero.

As for the power supply requirements of this ideal amplifier, there would be none. As you know, industry trends are driven by requests for lower supply voltages and, consequently, lower power consumption from active components. The ideal amplifier wouldn't need a voltage supply across V_{DD} and V_{SS} and would have zero power dissipation in its quiescent state.

The output of this amplifier would have no voltage limitations. This would eliminate the problem of losing bits on the outer rim in the subsequent A/D conversion as a result of the amplifier not being able to swing all the way to the rails.

The output impedance would be zero at DC and over frequency, ensuring that the external input device connected to the amplifier is isolated from the external output device. The op-amp would respond to input signals instantaneously (i.e., the slew rate would be infinite and there would be no delay), and it would be able to drive any load while maintaining an infinite open-loop gain and rail-to-rail output swing. Finally, in the frequency domain, the open-loop gain would be infinite at DC as well as over frequency, and the bandwidth of the amplifier would also be infinite.

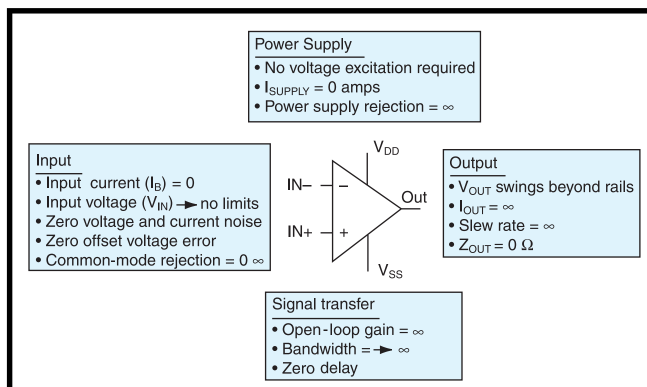


Figure 1—*A perfect amplifier has an infinite input impedance, open-loop gain, power supply rejection ratio, common-mode rejection ratio, bandwidth, slew rate, and output current. It also has zero offset voltage, input noise, output impedance, power dissipation, and most importantly, zero cost.*

We all want this ideal amplifier for free, however, if I design this amplifier, I guarantee I'll be a billionaire.

Interestingly, many of these design imperfections can be advantages. For instance, the less than infinite bandwidth of an amplifier is used to limit the noise and high-speed transients in circuits. This becomes an issue where circuit board traces would operate as transmission lines with reflections and such if the amplifier didn't perform this band-limiting function.

Today, there isn't an ideal amplifier for all circuit situations. The best you can do is pick the best amplifier for the application and use it properly.

CHOOSE WISELY

Single-supply operational amplifiers are commonly manufactured with CMOS and bipolar silicon technologies. It's usual for bipolar amplifiers to have bipolar input devices followed by CMOS transistors. This type is often called BiCMOS. The most important difference between CMOS and bipolar (or BiCMOS) is in the input stage transistors. These transistors have a profound effect on the operation of the amplifier (see Figure 2).

Because of the difference between the amplifiers' input transistors, the CMOS amplifier has lower input current noise and higher input impedance. As a consequence of the high-input impedance, the CMOS amplifier's input bias current is lower. In fact, the input bias current of a CMOS amplifier would be zero if it weren't for the ESD cells that are connected to the input pins. This can be used to an advantage for high-impedance sources.

The CMOS amplifier typically has higher open-loop gain than bipolar amplifiers. This can minimize gain error in applications where the closed-loop gain is high (60 dB or greater).

To contrast the CMOS amplifier, the bipolar amplifier usually has lower input voltage noise and offset voltage. Although these specifications are typically better than the CMOS amplifier counterpart, the input bias current and input current noise are higher.

Both CMOS and bipolar amplifiers can be designed for single-supply operation. If they are designed properly, they also are capable of input and output rail-to-rail operation.

BAD DATA IN, BAD DATA OUT

A signal transmission through the operational amplifier begins at the input stage. When selecting an amplifier, first scrutinize the characteristics of the external input signal.

For instance, what voltage range would you expect your external source signal to span? If the voltage range of this signal spans from one power supply rail to the other, a rail-to-rail input amplifier would be appropriate for your application. If not, you probably don't want a rail-to-rail input amplifier because there is an offset-voltage distortion that occurs as you take the input across its entire common-mode range (see Figure 3).

If this offset distortion feature is not desirable, you may want to consider designing your amplifier circuit in an inverting gain configuration. An example of this type of circuit is shown in Figure 4.

But, don't let this offset distortion scare you away if you really need rail-to-rail inputs. With single-supply circuits, rail-to-rail input amplifiers are needed when a buffer amplifier circuit is used or possibly with an instrumentation amplifier configuration. Be aware that if either of the inputs of the amplifier goes beyond the specified input range of that amplifier, the output will typically go to one of the power supply rails. There is no guarantee which rail.

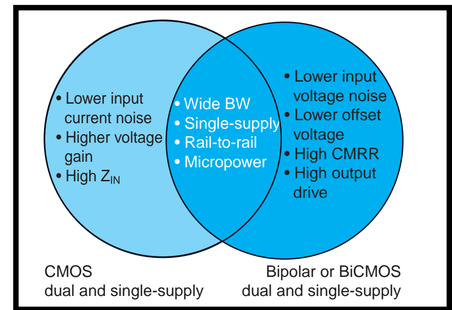


Figure 2—*Single-supply amplifiers are manufactured with CMOS and bipolar technologies. Sometimes bipolars are manufactured on a CMOS process, the input transistors are bipolar, and the rest are CMOS.*

HIGH INPUT IMPEDANCE MATTERS

The typical input bias current of a single-supply bipolar amplifier ranges from a few nanoamperes to hundreds of nanoamperes over temperature. Typical CMOS amplifier input bias current ranges from a few to hundreds of picoamperes. The effect of the error introduced by the input bias current depends on the magnitude of the source resistance and the circuit gain. Two examples of circuits that will be affected by high input bias current are shown in Figure 5.

The circuit shown in Figure 5 is designed to convert the light energy that impinges on the photo diode (D_P) into charge (or current over time). The current from the photo diode flows through the feedback resistor (R_F), generating a voltage at the output of A1. The output of A1 is directly connected to R_1, which is a part of a second order low pass gain/filter stage. This stage is built using A2, R_1, R_2, R_3, R_4, C_1, and C_2. Then, the output of the filter is sent to a 12-bit A/D converter.

If the two amplifiers in this circuit design are bipolar, the high-input bias

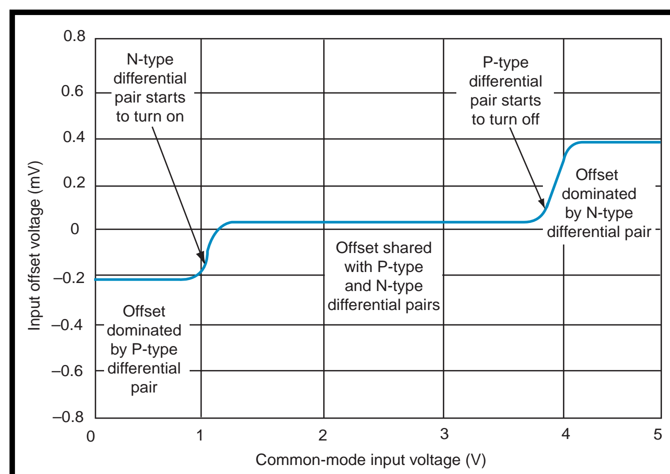


Figure 3—*Rail-to-rail input amplifiers require two pairs of differential transistors. They have an input-offset distortion as the input common-mode voltage passes through the regions where one pair is turning on or off. This amplifier has two offset-voltage transition regions; some single-supply amplifiers only have one transition region.*

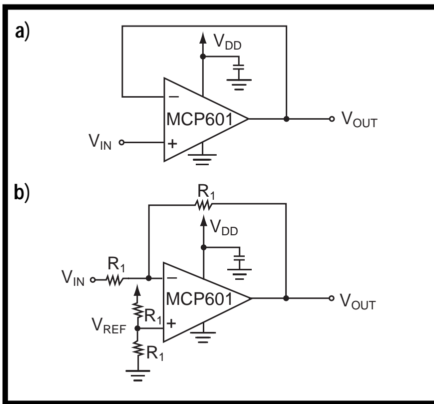


Figure 4—*If the input voltage swing is rail-to-rail, an amplifier configured as a single-supply buffer or voltage follower (**a**) should have rail-to-rail input capability. However, if an amplifier is configured in a gain of −1V/V (**b**), the amplifier input will remain at V_{REF}.*

current from the amplifiers can cause voltage errors in both stages. In the photo detection circuit the amplifier input bias current (A1) generates a voltage drop across the parallel combination of R_F and the photo diode parasitic resistance.

For example, assume the photo diode parasitic resistance and feedback resistance (R_F) are equal to 5 MΩ and 250 kΩ, respectively, and the input bias current of A1 is equal to 100 nA. The resulting voltage error from this combination is 23 mV, which appears at the output of the amplifier. In the anti-aliasing filter circuit, an input bias current can generate a voltage error across the input resistors (R_1 and R_2) at DC, which is amplified by the combination of R_3 and R_4 around the amplifier.

To continue this example, using the resistor values for R_1 and R_2 equal to 12.9 kΩ and 595 kΩ, respectively, and an input bias current from the amplifier of 100 nA, the resultant voltage error is 61 mV. This added to 23 mV from the previous stage, equals 84 mV of error. This voltage is multiplied by 10 V/V to equal a significant error of 840 mV at the input of the 12-bit A/D converter. If the LSB size of the 12-bit A/D converter is 1.22 mV, this error will produce a 688 count offset error.

This example can be recalculated using CMOS amplifiers with an input bias current of 60 pA (over temperature) instead of the bipolar amplifier selected above. With this new amplifier, the voltage error generated in the first stage is 14 µV. The voltage error generated as a consequence of the two resistors in the anti-aliasing filter stage is 36.5 µV.

This total error, multiplied by 10 is equal to 508 µV, which is presented to the 12-bit A/D converter input. Now, with an A/D converter LSB size of 1.22 mV, the error from the analog front end produces a 0.416 bit error.

REALLY RAIL-TO-RAIL?

Single-supply amplifiers do not truly swing rail-to-rail at the output. To make matters worse, at the outer regions (near the rail), the amplifier will behave in a non-linear fashion. The reality of this performance characteristic is that the output of single-supply amplifiers can only come within 50 to 200 mV of each rail. This behavior is illustrated in Figure 6.

In advertising, the claim of "rail-to-rail" can give you a false sense of security, meaning the amplifier will operate as an amplifier for the full output range. Figure 6 illustrates what the output swing of a single supply amplifier looks like when the output is driven to the rails.

In Figure 6, notice that the linearity of the amplifier starts to degrade before the output swing maxima are reached. If the amplifier output is operated beyond the linear region of this curve, the signal's input-to-output relationship will be nonlinear.

The conditions of the DC open-loop gain (A_{OL}) specification defines the linear operating output range of the amplifier. The definition of DC open-loop gain (A_{OL}) is:

$$A_{OL}\ (\mathrm{dB}) = 20 \log (\Delta V_{OUT} / \Delta V_{IN})$$

where V_{OUT} is the output voltage and V_{IN} is the input offset voltage. ΔV_{OUT} is $(V_{OH} - V_{OL})$, where V_{OH} is the maximum voltage level of the output when it is driven high and V_{OL} is the minimum voltage level when it is driven low. And, ΔV_{IN} is the range of input voltage that produces this change in output voltage.

The difference between the A_{OL} conditions and an output swing-limited condition are profoundly different. The A_{OL} specification validates the voltage output swing test by implying that the operational amplifier is operating within its linear region. But, taking this discussion beyond the difference of these specifications, the output of your single-supply amplifier will never reach the power supply rail. Design your circuits accordingly.

DO YOU HAVE THE BANDWIDTH?

The gain bandwidth product (GBWP) of an operational amplifier is equal to the frequency of the first pole of the open-loop gain plot multiplied by the gain at that first pole. Most operational amplifiers are unity-gain stable, so the GBWP happens to be equal to the 0-dB crossing.

You will seldom find an amplifier that has a guaranteed gain bandwidth product. Usually, the GBWP of an operational amplifier is specified as a typical and varies ±20 to ±30%. This type of variation is not a good specification to hang your hat on. Consequently, most designers use resistors and capacitors to control the bandwidth of the amplifier circuit. A common circuit that this is done with is shown in Figure 7. This 5th order, low-pass filter is designed to limit the bandwidth of a system (i.e., prevent aliasing errors). This is only an example of a 5th order filter. It could be a 2nd, 3rd, 4th,…order filter.

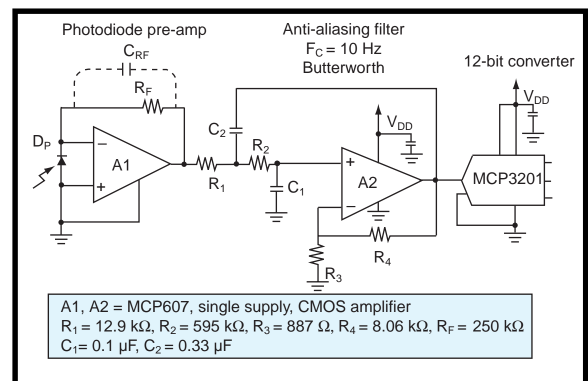


Figure 5—*In photo detection circuits, the amplifier input bias current can generate voltage drop across the parallel combination of R_F and the photodiode parasitic resistance. In anti-aliasing filter circuits, input bias current can generate a voltage error across input resistors R_1 and R_2.*

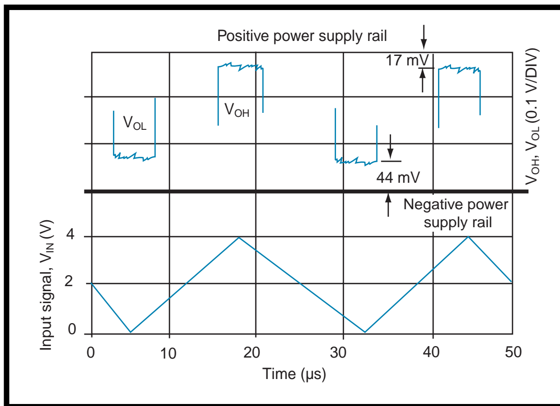


Figure 6—*The bottom plot illustrates the input voltage swing to an amplifier that is configured in a gain of +2V/V. The top plot shows the magnified output voltage of the amplifier. Included in the top portion of this plot is an indication where the positive supply rail is with respect to VOH and where the negative supply rail is with respect to V_{OL}.*

When designing this circuit, you should consider the bandwidth of the operational amplifier. If you are too close in frequency to the operational amplifier bandwidth, the amplifier will contribute another pole to the transfer function. This will more than likely cause instability in the circuit.

Taking it further, you may have designed an amplifier for a gain of 10 and find that the AC output signal is lower than expected. If this is the case, you may have to look for an amplifier with a wider bandwidth. A general rule of thumb is to design the cutoff frequency of your filter at least 10 times lower than the closed-loop crossing of the amplifier's open-loop gain curve.

DON'T MISS THE DETAILS

Analog design experience is available through many channels. One of my favorite channels is hands-on. Hands-on engineers are the people with scar tissuse covering their fingertips caused by years of looking at problem circuits, asking what went wrong, and then touching every chip.

Another tell-tale sign of experience is when a designer has dead bug circuits in his workspace. This type of circuit is built by turning the device on its back, with pins sticking straight up, and soldering all connections to the device in the air. Using this technique is an attempt to eliminate parasitic capacitance.

Yet another indication that an individual spends time in the lab is facial scar tissue that looks like little razor cuts. This may indicate a shaving problem, but more than likely the designer was looking at a circuit when a capacitor or chip "blew up."

Knowledge comes with these kinds of experiences. Here are a few tips. Before you plug the chip in, double check your power supplies.

If the positive supply is too high at any time with respect to the negative supply, the part will likely be damaged. In contrast, a low supply won't bias the amplifier's internal transistors. A simple check of the difference between the supply voltages at the pins of the operational amplifier can save a great deal of troubleshooting time. A note of caution, turn the supplies off before you insert the operational amplifier in the socket.

Double-check your grounding strategy, especially if there are digital circuits on the board. Low-impedance grounds are imperative if you want a stable analog design. If the circuit has a lot of digital circuitry, consider separate ground and power planes. Ground noise is challenging to track because it appears everywhere.

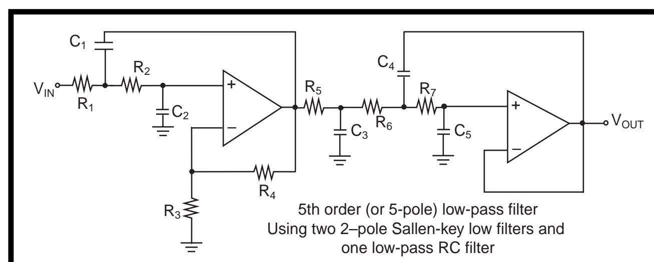


Figure 7—*A low pass filter, such as this one, typically is used to remove high frequency noise. The poles of this filter are carefully designed using the R/C relationships in the circuit. If the cut-off frequency of the filter is too close to the closed-loop gain curve of the amplifier, this circuit may oscillate.*

Always decouple the amplifier power-supply pins with capacitors. Place these capacitors as close to the amplifier pin as possible. For amplifiers with a bandwidth up to tens of megahertz, a 1-µF or 0.1-µF capacitor is recommended.

Breadboarding on white breadboarding sockets is a risky way of doing circuit evaluation. These boards can produce noise or oscillations because of the preponderance of capacitance and inductance underneath the board. Because you should use short lead lengths to the inputs of the amplifier the perf board will fail you. These, may not be a problem with the PCB implementation of the circuit.

Amplifiers are static sensitive! If they are damaged, they may fail immediately or exhibit a soft error that will continue deteriorating over time.

I discussed the common problems with op-amp design implementation. If you have other inputs from experience, feel free to drop me a note. ■

Bonnie Baker is a staff applications engineer at Microchip Technology. Formerly a music teacher, since earning her MSEE she has held a variety of positions, including analog design engineer, applications engineering manager, and strategic marketer. Bonnie has prepared and presented seminars and published numerous articles in technical magazines.

REFERENCES

S. Franco, *Design with Operational Amplifiers and Analog Integrated Circuits*, McGraw Hill, New York, NY, 1998.

T. Frederiksen, *Intuitive Operational Amplifiers*, McGraw Hill, New York, NY, 1988.

J. Williams, *Analog Circuit Design*, Butterworth-Heinemann, Stoneham, MA, 1991.

B. Baker, "Anti-aliasing Analog Filters for Data Acquisition Systems," AN699, Microchip Technology Inc.

B. Baker, "Using Single Supply Amplifiers in Imbedded Systems," AN682, Microchip Technology Inc.

B. Baker, "Operational Amplifier AC Specifications and Applications," AN723, Microchip Technology Inc.

FEATURE ARTICLE

Mel Bartels

Keep Up with the Stars

Motorized Telescope Control

Looking at the stars is one thing, but according to Mel, tracking them is much more interesting, which is why he designed a computer-controlled motorized telescope mount. Build this project and you'll be seeing stars in no time at all.

When 286-level processors were introduced, sky watching became clearer. Although microprocessors were available, they weren't practical for many astronomy enthusiasts. In this article, I'll cover telescope basics and then head into controlling a motorized telescope with your computer to get the best results.

The popular Newtonian telescope uses a paraboloidal mirror to focus light onto an image plane that is then inspected by an eyepiece to gain magnification. The paraboloidal mirror and smaller flat secondary that is used to direct light out to the side of the tube must be ground, polished, and figured to an accuracy of a couple millionths of an inch. Amateurs can do this by hand using simple test equipment and can achieve an astonishing degree of penetration into the sky. Using a CCD camera and 12″-aperture telescope, an amateur can equal the 200″-Palomar telescope with film plates. Digital cameras with cooled detectors must be exposed for many minutes to capture faint objects. During the exposure, the camera and telescope must precisely follow the stars moving slowly.

Photo 1 shows my 20″-aperture computer-controlled telescope. Photo 2 shows a CCD camera exposure of Messier 13, The Great Hercules Globular Star Cluster.

The challenge of motorizing mounts with large thin mirrors causes most amateurs to build hand-pushed telescopes. The Dobsonian mount relies on Teflon and Formica surfaces to give smooth hand motion, yet remain stationary when released. Coma is a comet-shaped optical aberration caused by the paraboloidal primary. It is observed when viewing objects off-axis. Today's fast f/5 telescope produces coma exceeding 1/4 wavefront a few arcminutes off center. A planet drifts this distance in a few seconds in the viewfinder of this telescope. Experienced observers must wait for fleeting moments of clear vision when the atmosphere steadies and good planetary detail bursts through. Motorized tracking means using these moments while keeping the object precisely centered in the field of view.

The tracking accuracy of the telescope mount is particularly demanding. With respect to the stars, earth

Photo 1—*A 20″-aperture computer-controlled telescope is shown here.*

rotates once every 23 h, 56 min. (earth overshoots every day by 1/365 of a rotation). This is called a sidereal day. The telescope must move at this rate, referred to as the sidereal tracking rate. Earth's atmosphere limits resolution usually to a couple of arcseconds (about 1/2000 of a degree). At the best sites, resolution rarely exceeds 1 arcsecond. Consequently, in order to produce round star images, your tracking system needs to be accurate to 1/4 arcsecond. If it's accurate to one part in 2^{16}, you can take unguided long exposure images on either film or silicon, up to 20 min.

Atmospheric refraction calls for slower tracking rates near the horizon. Refraction, dependent on barometric pressure and relative humidity, makes objects appear higher than usual in the sky. At the horizon, refraction reaches an angle of 34 arcminutes. Objects like the moon need different tracking rates.

The best amateur portable tracking telescopes achieve 30 s to 2 min. unguided, and permanently mounted professional telescopes reach 20 min. The principal limiting factor is error in the gearing. Worm and gear errors include periodic and erratic errors. The gear's elliptical shape and off centering of the worm on its shaft cause periodic errors. Tooth-to-tooth differences and backlash when the drive changes direction cause erratic errors. By using a gear reducer in the preliminary stage and a roller drive for the final stage, the errors in the former are divided by the ratio of the final roller. For example, if the gear reducer has a 1 arcminute error, and the final roller drive ratio is 30:1, the error at the eyepiece is 2 arcseconds.

Many professional scopes use circular rollers driven by machine shafts. These avoid the errors in worm and gear drives. Exposures using automated guided systems to follow a nearby star can extend longer. Modern telescope control systems capture guiding corrections and later play them back mirrored to compensate for periodic error correction, called PEC.

A PEC table is built with 200 entries, which cover any number of full steps. Periodic error will likely repeat every 200 full steps, corresponding to a single revolution of the stepper and the attached worm. If the worm is a double, quad, or other and gears downstream need periodic correction, the periodic error can occur over more full steps. Error values can be linearly interpolated between entries.

If one of the telescope mount's axes is pointed at the celestial pole, only that motor needs to turn at a constant rate (see Photo 3). A crystal-stabilized motor-control circuit is usually used to achieve precise tracking rates despite varying temperatures during the evening (see Photo 4).

If the mounting is altazimuth, which operates like a gun turret with spin to the left, right, up, and down, both axes must be driven at slowly varying rates (see Photo 5). As the star moves, it scribes an arc across the sky. In the past, unless you could afford a minicomputer, updating drive rates on the fly while controlling the motors was a dream until 286-level processors were introduced. The 6502 processor couldn't update frequently enough to achieve smooth tracking.

To achieve smooth motion, command so that start and stop motions are not perceptible. Analogous to the monitor's refresh rate, a 30-Hz command or update frequency rate is sufficient. The mount's fundamental harmonic frequency also helps. Most amateur mounts have oscillatory frequencies of several hertz and are only moderately stiff. Higher frequencies are attenuated and smoothed out by the mount.

It's challenging to achieve 16-bit control. Servo motors with velocity feedback are flexible, but the digital encoder feedback is expensive. Stepper motors with positional feedback achieve finer control at slower speeds but suffer from coarse discreet steps, vibration, and limited high speed. Both systems have several successful installations, but servo motors work better for equatorial telescopes and stepper systems work better for inexpensive altazimuth mounts.

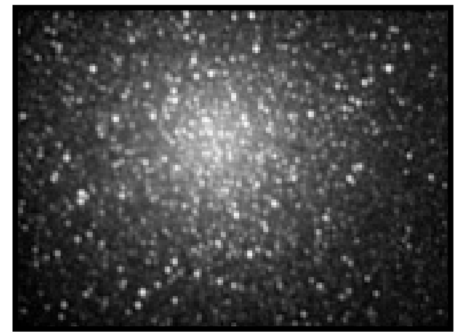


Photo 2—*Here's what Messier 13, the Great Hercules Globular Star Cluster, looks like in a CCD camera exposure.*

Sequence of control words output (10 pulses per phase):

Phase 1: 1 1 1 1 1 1 1 1 1 1
Phase 2: 2 2 2 2 2 2 2 2 2 2
For full stepping at half current:
Sequence of control words output (10 pulses per phase):

Phase 1: 1 1 1 1 1 0 0 0 0 0
Phase 2: 2 2 2 2 2 0 0 0 0 0
For half stepping at half current where the intermediate half step consists of both winding A and B on:
Sequence of control words output (10 pulses per phase):

Phase 1: 1 1 1 1 1 0 0 0 0 0
Phase 2: 3 3 3 3 3 0 0 0 0 0
Phase 3: 2 2 2 2 2 0 0 0 0 0

Table 1—*This shows the resulting sequence of control words for a single full step with maximum average current (ignoring the other windings on bits #2 through #7) if bit #0 controls winding A (control word output = 1) and the control word bit #1 controls winding B (control word output = 2).*

Sequence of control words output (10 pulses per phase):
----
a) Winding A at 100% current: 1 1 1 1 1 1 1 1 1 1
+ Winding B at 60% current: 2 2 2 2 2 2 0 0 0 0
= Winding A + winding B: 3 3 3 3 3 3 1 1 1 1
Therefore, to microstep with four microsteps per full step with maximum current:
b) Sequence of control words output (10 pulses per phase):

Phase 1: 1 1 1 1 1 1 1 1 1 1 (A current = 100%, B current = 0%)
Phase 2: 3 3 3 3 3 3 1 1 1 1 (A current = 100%, B current = 60%)
Phase 3: 3 3 3 3 3 3 3 3 3 3 (A current = 100%, B current = 100%)
Phase 4: 3 3 3 3 3 3 2 2 2 2 (A current = 60%, B current = 100%)
Phase 5: 2 2 2 2 2 2 2 2 2 2 (A current = 0%, B current = 100%)

Table 2—*Because electromagnetic fields propagate as the inverse square, the current supplied to B must be about 60% of A's current.*

Stepper motors are limited at high speeds. As the computer switches current on/off to the windings, counter electromotive force (EMF) is generated. When the current source is switched off, the collapsing magnetic field moving quickly through the winding generates a voltage spike that can destroy the power transistors.

A flyback diode prevents the voltage spikes by giving a path for the dying current to circulate back into the winding. However, this slows the time for the current to collapse. As the motor tries to spin faster, torque decreases. A zener diode used with the flybacks allows only the voltage above the zener's rating to be returned to the power source. This prevents extreme voltage spiking and avoids full braking of the flyback diodes.

While using higher voltage than the motor's continuous voltage rating and smoothly raising the spin, you can achieve faster speeds. Rates up to 10,000 half steps per second can be achieved with modest torque. I use a 12-V battery to operate 6-V steppers, which gives enough voltage to run the steppers at high speed. A motor stepping at high speed consumes minimal current because little current can be forced into the motor.

The array holding the half stepping pattern for two motors, A and Z, is initialized by the code in Listing 1. Using the parallel port, this code energizes windings IxA and IxZ of each motor, respectively:

```
outportb( MotorPort,
 HsOut[IxA)].A +
 HsOut[(IxZ].Z);
```

where `MotorPort = *(unsigned far *)(MK_FP( 0x40, 0x008 + (lptnum-1)*2));`

The PC's parallel port can't source a lot of current, so power transistors control the stepper windings. Figure 1 shows how a single parallel port data output line controls one of the unipolar stepper motor's four windings.

The PC's timer chip can time the half steps using Listing 2's values. Reset the PC's time after the call to the CMOS RTC (see Listing 3).

Listing 1—*The array holding the half stepping pattern for two motors, A and Z, is initialized by the code shown here.*

```
HsOut[0].A = 1;
    HsOut[1].A = 3;
    HsOut[2].A = 2;
    HsOut[3].A = 6;
    HsOut[4].A = 4;
    HsOut[5].A = 12;
    HsOut[6].A = 8;
    HsOut[7].A = 9;
    for( IxA = 0; IxA < 8; IxA++)
            HsOut[IxA].Z = HsOut[IxA].A << 4;
```

MICROSTEPS

For smooth slow-speed motion, steppers are microstepped. With a PC controlling the stepper motor windings' voltage waveforms, you can divide full steps into microsteps. Microstepping gives five advantages:

- resonance frequencies allow good low-speed operation
- dynamic range with lower frequencies helps avoid ringing, noise, and vibration
- the gearbox is replaced by extending the number of steps per motor shaft revolution
- better step accuracy
- less system complexity

To microstep, winding A slowly ramps down while winding B slowly ramps up. Applying full current to winding A puts the rotor over it. Applying equal current to both windings puts the rotor between them. Applying a current to winding B that is 60% of winding A's current puts the rotor 1/4 of the way between them.

Listing 2—*The PC's timer chip can be used to time the half steps using these values.*

```
#define Timer_Int 8
    /* set new interrupt handler */
    disable();
    ClockVect = getvect( Timer_Int);
    setvect( Timer_Int, SlewTickHandler);
where
#define PIC_EOI_ADDR 0x20
#define EOI 0x20
void interrupt SlewTickHandler( void)
{
    if( !Done)
    {
            DoOneHs();
            SetTickDelay();
    }
    /* enable PIC EOI (end of interrupt) */
    outportb( PIC_EOI_ADDR, EOI);
}
and SetTickDelay( void)
has at its end:
    Timer = <calculated value between halfsteps>;
    /* set 8259 chip, Timer 0, read low, then high, software
triggered strobe */
    outportb( Timer_Control, 0x38);
    /* low byte */
    outportb( Timer_0, 0xFF & Timer);
    /* high byte */
    outportb( Timer_0, 0xFF & (Timer >> 8));
where
#define Timer_0 0x40
#define Timer_Control 0x43
```

Microstepping's limitations include deflection error caused by torque loading and absolute tooth error, typically 1/25 of a full step. The deflection error is minimal when the rotor is positioned on a winding and maximum when positioned between windings. If the torque loading is 10%, the shaft's error between windings will be 10% of a full step. Microstepping at 10 microsteps per full step is a reasonable compromise between smoothness and rotor position accuracy. More microsteps can mean a smoother motion, but will not increase rotor position accuracy.

The parallel port's eight output bits can simultaneously control the current waveform of the stepper motors' eight windings. The current waveforms are generated using pulse width modulation (PWM). Full current is turned on for a certain time then turned off. The cumulative effect of rapidly repeating on and off is the same as if smooth average current. By adjusting the percentage of on versus off, the resulting current can be controlled precisely. Torque remains high regardless of the motor's speed because full current is applied during the on time.

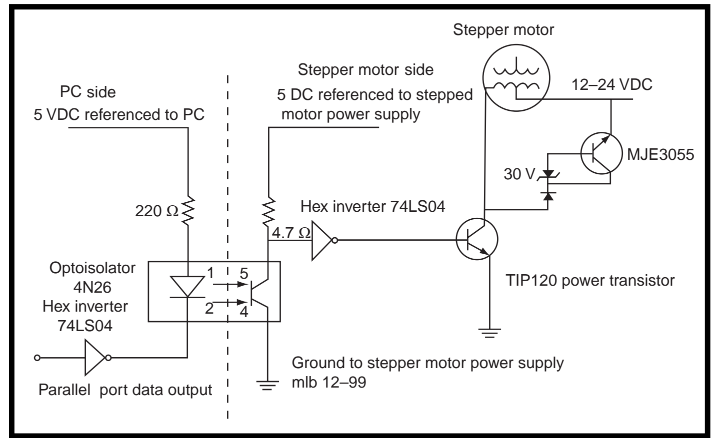


Figure 1—*This is how a single parallel port data output line controls one of the unipolar stepper motor's four windings.*

For adequate current resolution, the sequence of on and off will add up to 100 or more. Let's say that the total sequence per phase is 10. Table 1 shows the resulting sequence of control words for a single full step with maximum average current (ignoring the other windings on bits #2 through #7) if bit #0 controls winding A (control word output = 1) and the control word bit #1 controls winding B (control word output = 2).

Place the rotor at intermediate positions between windings A and B to microstep. To set the rotor 25% towards B, the rotor must feel winding B one-third as much, positioning itself three times closer to A than to B. Because electromagnetic fields propagate as the inverse square, the current supplied to B must be about 60% of A's current (see Table 2).

```
  0: 1 1 1 1 1 1 1 1 1 1
 10: 3 1 1 1 1 1 1 1 1 0
 20: 3 3 1 1 1 1 1 1 0 0
 30: 3 3 3 1 1 1 1 0 0 0
 40: 3 3 3 3 1 1 0 0 0 0
 50: 3 3 3 3 3 0 0 0 0 0
 60: 3 3 3 3 2 2 0 0 0 0
 70: 3 3 3 2 2 2 2 0 0 0
 80: 3 3 2 2 2 2 2 2 0 0
 90: 3 2 2 2 2 2 2 2 2 0
100: 2 2 2 2 2 2 2 2 2 2
110: 6 2 2 2 2 2 2 2 2 0
120: 6 6 2 2 2 2 2 2 0 0
130: 6 6 6 2 2 2 2 0 0 0
140: 6 6 6 6 2 2 0 0 0 0
150: 6 6 6 6 6 0 0 0 0 0
160: 6 6 6 6 4 4 0 0 0 0
170: 6 6 6 4 4 4 4 0 0 0
180: 6 6 4 4 4 4 4 4 0 0
190: 6 4 4 4 4 4 4 4 4 0
200: 4 4 4 4 4 4 4 4 4 4
210: 12 4 4 4 4 4 4 4 4 0
220: 12 12 4 4 4 4 4 4 0 0
230: 12 12 12 4 4 4 4 0 0 0
240: 12 12 12 12 4 4 0 0 0 0
250: 12 12 12 12 12 0 0 0 0 0
260: 12 12 12 12 8 8 0 0 0 0
270: 12 12 12 8 8 8 8 0 0 0
280: 12 12 8 8 8 8 8 8 0 0
290: 12 8 8 8 8 8 8 8 8 0
300: 8 8 8 8 8 8 8 8 8 8
310: 9 8 8 8 8 8 8 8 8 0
320: 9 9 8 8 8 8 8 8 0 0
330: 9 9 9 8 8 8 8 0 0 0
340: 9 9 9 9 8 8 0 0 0 0
350: 9 9 9 9 9 0 0 0 0 0
360: 9 9 9 9 1 1 0 0 0 0
370: 9 9 9 1 1 1 1 0 0 0
380: 9 9 1 1 1 1 1 1 0 0
390: 9 1 1 1 1 1 1 1 1 0
```

Table 3—*A microstepping array is created and filled with the PWM values so that during program execution, only writing each array element in turn to the port is necessary.*

A microstepping array is created and filled with the PWM values so that during program execution, only writing each array element in turn to the port is necessary. For example, if the 10 microstepping PWM values are 10, 9, 8, 7, 6, 5, 4, 3, 2, and 1, the resulting array will have 400 entries (10 PWM per microstep × 10 microsteps per full step × 4 full step windings per sequence). That makes 40 microsteps (10 microsteps per full step × 4 full step windings per sequence), as shown in Table 3.

Tweaking the PWM values adjusts for the finite on/off times of the power transistors, hex inverters, optoisolators, parallel port, differences in speed between PCs, and differences between motors and torque loading.

Motors work best with 300 and 2000 per second PWM counts. If you use the PC's BIOS clock tick as a convenient time marker, each tick should have between ~20 and ~100 PWMs. Even if a stepper motor is stationary, it is necessary to send PWMs to the motor to maintain its rotor position. If the current is turned off, the rotor will move to the nearest full step winding, thanks to the permanent magnetism of the motor. These detents can be felt by slowly spinning the shaft of a disconnected motor by hand.

BIOS CLOCK TICK

Whether using velocity-based servo systems or position-based stepper systems, an accurate timing signal must be procured. A PC's BIOS clock tick does this. An altazimuth-mounted telescope moves in a line between each clock tick. The deviation between this line and the arc a star will follow between clock ticks cannot exceed an appreciable fraction of an arcsecond. The BIOS timing tick happens often enough, 18.2 times per second, to alleviate this worry.

The sequence of events for each BIOS clock tick begins with adding equatorial drift to current equatorial position. Next, update a status field or work with the optional encoders. Then, perform the following series of checks if the previous one doesn't occur. For example, check for a keyboard event–if there isn't one, check for a hand paddle event, and so on. The next steps are:

- check for an IACA (inter-applications communication area) event, if none,
- check for LX200 protocol serial port events and process all accumulated commands since the last BIOS clock tick, if none,
- check if field rotation motor needs pulsing

Then, move to current equatorial coordinates by calculating new altazimuth coordinates based on the new sidereal time that was calculated when the BIOS clock tick occurred. Calculate the difference between current altazimuth coordinates and newly calculated altazimuth coordinates. Determine the distances to move in each axis and, if you're microstepping, choose microstepping or half stepping.

If there isn't backlash, spread microsteps over the BIOS clock tick by dividing the number of microsteps into MsTicksRep, the count of PWMs per BIOS. If microsteps exceed MsTicksRep, reduce the number of microsteps per full step to half step.

Continuously generate PWMs, checking for a BIOS clock tick at the end of each PWM. Next, port an already calculated array of on and off to the stepper motors' windings.

Listing 3—*The PC's time needs to be reset after the call to the CMOS RTC.*

```
   Flag ReadRealTimeClock( int* hr, int* min, int* sec, int* day-
light)
   {
       Regs.h.ah = 2;
       int86( TimeOfDayInterrupt, &Regs, &Regs);
       BCD = Regs.h.ch;
       *hr = DECODE_BCD;
       BCD = Regs.h.cl;
       *min = DECODE_BCD;
       BCD = Regs.h.dh;
       *sec = DECODE_BCD;
       *daylight = Regs.h.dl;
       /* carry flag set if RTC not running (inaccessible) */
       return !Regs.x.cflag;
   }
   where #define DECODE_BCD (int) (((BCD & 0xF0)>>4)*10 + (BCD &
0x0F))
```

When BIOS clock tick occurs, PWMs end, new sidereal time is calculated, and current altitude coordinate is updated. To include refraction, current altazimuth coordinates are updated to include backlash compensations that already moved. And, the coordinates are updated to reflect the number of microsteps that occurred and to include PEC based on the stepper rotors' position, altazimuth drift, and guiding motions.

The steppers operate open loop. If the stepper motors never stall and the scope isn't bumped, the software always knows the scope's aim. After a bump, the computerized finding breaks, and the tracking becomes poor because the drive rates vary for each sky position. Center the scope on a known object and inform the software.

As recourse, you can use encoders, 2', inexpensive devices that convert rotary motion into digital pulses. The pulses are counted by a microprocessor. The PC queries the microprocessor via a RS-232 serial connection for the current counts, converting the counts to shaft angles. The parallel port is sending output to the stepper motors and receiving control signals from the hand paddle, and the serial port is communicating with the encoder interface. Using the encoders within a mouse is another alternative. No interface box is necessary, simply load a mouse drive with acceleration and multiplier options set to off.

Photo 3—*If one of the telescope mount's axes is pointed at the celestial pole, only that motor needs to turn at a constant rate.*

The popular incremental optical shaft encoder consists of an optical disk with alternating clear and opaque spokes. Two LEDs shine onto detectors through the spokes; they are staggered so that when the optical disk rotates, the following sequence occurs:

```
                  time --->
outer detector: on  off on off
inner detector:    on  off on off
```

If the disk rotates the opposite direction, the sequence occurs backwards:

```
                 time--->
outer detector:   on  off  on  off
inner detector: on  off  on off
```

The microprocessor decodes each passing of a spoke into four events, known as quadrature decoding.

The microprocessor must handle the encoder's pulse train speed. If the encoder is geared 8:1 and the shaft spins at one rev per second, the total events from the outer detector (A channel) and inner (B channel) occur at 64 kbps. To reject noise, the processor can sample each channel three times, accepting the result only if all reads are the same. And, it must count while communicating with the PC.

In a permanently-mounted telescope, the encoder interface can stay on always and be run by a small battery. When the PC is off, the encoder interface counts encoder pulses, always knowing the scope's aim. In an altazimuth mount, only the tube assembly in altitude must be set to a known angle at start-up. The Taki routines do not need to synchronize the starting azimuth value to the scope's starting direction.

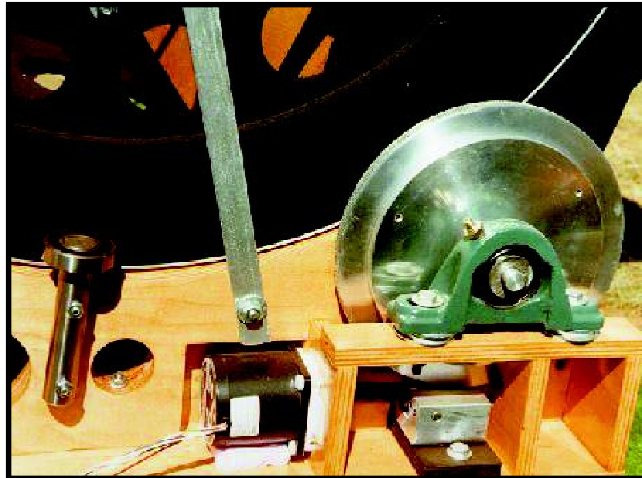


Photo 4—*A crystal-stabilized motor-control circuit is usually used to achieve precise tracking rates despite widely varying temperatures in the evening.*

WRITING CODE

Writing code for real-time control is both a bottom–up and top–down process. Port I/O and simple motor commands are built from the bottom up. Working from the bottom up, `DoOneHs()` causes the stepper motors to move one half step. Consequently, it's called by `SlewTickHandler()`s, the interrupt handler for the PC timer chip that is installed by the function `MoveHsUsingIRQTimer()`. This function is called by `MoveHs()`.

At the highest level, sequences of objects can be automated so the operator can press a button to view the next object. High magnification, smooth scrolling tours that last minutes can be flown over large objects, returning to the star position, ready for the next person at public star gazing parties.

Working from the top down, `main()` calls `ProcessKmwBEvents()` (keyboard), which calls `ProcessHPEvents()` (hand paddle), which calls `MoveToCurrentRaDec()`, which calls `SetDirDistanceStepsThenMove()`, which calls

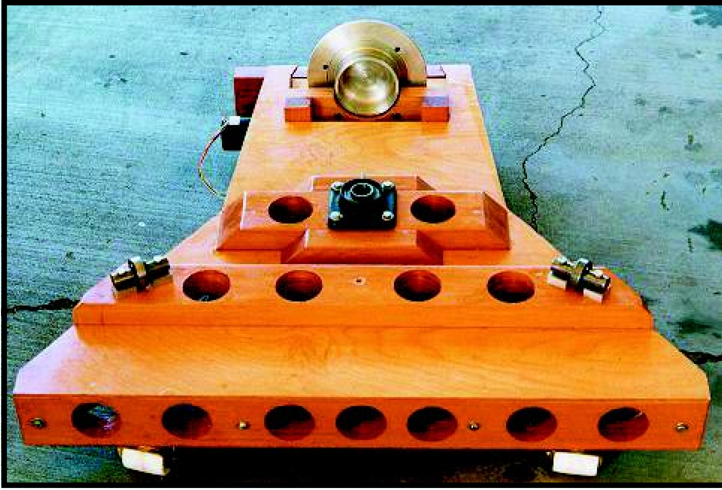


Photo 5—*If the mounting is altazimuth, which operates like a gun turret with spin to the left, right, up, and down, both axes must be driven at slowly varying rates.*

`KBEventMoveHs()`, which calls `HPEventMoveHs()`, which calls `MoveHs()`, jumping into the lower-level functions.

The following inaugurate high-level sequences that involve calculations, real-time screen updates, and lower-level motor control function calls: keyboard events, hand paddle events (read via the parallel port) events , commands assembled from a serial input port, and files containing agreed on formats that show a new time or date stamp. If there are no events, the software continues tracking the currently targeted object.

It is possible to use astronomical markup language (AML) to send commands via e-mail, which are translated into LX200 protocol serial commands. A Linux box with a Perl script can parse the e-mail and send commands out the serial port to a DOS box running the control program. Similarly, images can be retrieved from a separate PC running the CCD camera and e-mailed to the operator.

Software and related information is offered for free on the Internet. It was gratifying to receive significant contributions from others around the world, making my system better than I could achieve myself. ■

A former orchestral musician and teacher, Mel Bartels is now a programmer and systems manager. He also builds computer control systems for telescopes and runs a web site for making amateur telescopes. His web page is at zebu.uoregon.edu/~mbartels/altaz/altaz.html.

REFERENCES

Johnson, *Build Your Own Low-Cost Data Acquisition and Display Devices*, Tab Books, 1994

Trueblood and Genet, *Microcomputer Control of Telescopes,* Willman-Bell, 1997

FEATURE ARTICLE

Brian Millier

Back to the BasicX

Part 1: NetMedia's Development System

If you're a creature of habit when it comes to programming languages, you may want to listen to what Brian has to say. With a design project at hand, he realized it was time to break the shackles of tradition and take a look at NetMedia's BasicX.

What does it take to make a dyed-in-the-wool assembly language programmer consider a high-level language? After 20 years of designing circuits with numerous microcontroller families using assembly language firmware, I thought nothing could convince me to change. I preferred having nothing separating my source code from the internal CPU core and any external peripherals. If something doesn't work as expected, I simply look at the datasheets and my own program code.

My second rule is to use the smallest possible microcontroller, preferably one with all of the required memory onboard that needs few external peripheral devices. Although few of my designs could be considered high speed, usually there is a time-critical aspect of the design. For all of these reasons, a design based on firmware written in a high-level language has always been ruled out.

My last few projects involved Microchip's high-speed PIC family of microcontrollers. Although the designs worked, the assembly language programming, with limited RISC instruction set and addressing modes, was enough to make me take another look at higher-level languages.

Lest you think I'm stuck in the '80s, I stopped writing assembly language programs for any PC-based applications when Windows 3.1 became popular. I liked QuickBasic for DOS, so Visual BASIC became my programming for Windows.

BASICX DEVELOPMENT SYSTEM

I recently saw advertisements for NetMedia's BasicX microcontroller chip and associated development system. They claim to execute high-speed BASIC programs, as well as built-in high-speed networking and multitasking. Because I had a design project on the horizon involving devices best interconnected through a network topology, I decided to investigate BasicX. The increasingly commercial nature of the world wide web frustrates me, but the ability to download varied information within half an hour is rewarding!

My first impression of the BasicX system was positive. Although the ads didn't mention what processor was used in this product, I discovered it was an Atmel AVR chip. I downloaded its datasheet and found it to be a fast chip with a great instruction set, lots of onboard flash memory, EEPROM, and SRAM, as well as

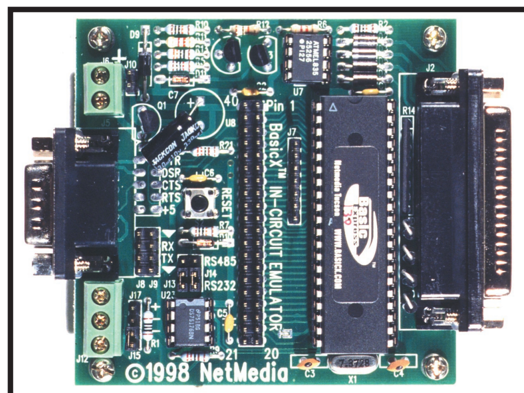


Photo 1—*The BasicX development board/programmer is only 3" × 3". The 25-pin D socket connects to the host PC's printer port, which controls the board and supplies it with power. Mounted on the opposite side are the serial and network ports, as well as an external power socket.*

splendid peripheral functions. If I had to live with the slower execution speed of BASIC firmware, this seemed like a fast engine on which to run it.

The BasicX development board/ programmer is a small board that connects to the printer port of a PC (see Photo 1). For $50, you can buy this board, including a CD-ROM containing the Compiler/IDE software and documentation. Two cables are also included. The first cable allows you to connect the development board to the target board (in place of the BasicX chip) for full speed testing of your code, and the second allows you to download the program code directly to the target board, using a simple seven-wire connection.

The development system has a simple application preprogrammed into its EEPROM memory. By powering the BasicX board and connecting its serial port to a free COM port on your host PC, you can verify that the processor works by watching a message that is sent once every second. On the units I received so far, this was either a `hello world` message or a continually updated time and chip serial number. It's always good to see that something has been tested and is working, early in the game!

THE HARDWARE

Being a hardware person at heart, I can't resist first explaining what's involved in the BasicX hardware before going into detail about the features of the language, the operation of the compiler, and so on.

Figure 1 shows a typical small BasicX system including the network function and a serial port. This is similar to the circuitry on the BasicX development board itself, except the development board steals its power from the host printer port connection, resulting in a few circuit changes. Also shown in the Figure is the wiring diagram for a cable to connect it to a PC's printer port for programming (using NetMedia's BasicX compiler and downloader software).

The first point to note is the presence of an Atmel AT25256-SPI EEPROM memory chip connected to the Atmel 90S8515 microcontroller's SPI port. The BASIC pseudocode interpreter resides permanently in the 'S8515 flash memory array and the user's application code, after being compiled into pseudocode, is stored in the serial EEPROM memory. The AT25256 EEPROM is 32K × 8, and holds between 500 and 1000 lines of BASIC code. A simple seven-pin header, connected to the SPI EEPROM and the microcontroller *RESET line, constitutes the programming interface to the host PC. Although the 'S8515 microcontroller supports a clock rate up to 8 MHz, a 7.3728-MHz crystal is used to provide exact UART data transfer rates, as well as to simplify the real-time clock (RTC) design.

The network function uses the three-wire RS-485 standard with a single 75176 transceiver chip. An RS-485 link must be terminated at both ends of the run, so a 100-Ω resistor/ jumper is provided for the end units. RS-485 is a multidrop network that uses a differential pair of signals on its NET+ and NET– lines, so it works reliably over long distances and allows many devices to be connected to it. RS-485 is a half-duplex protocol that supports peer-to-peer networking, assuming that the software handles the inevitable data collisions. The RS-485 network uses a common ground line for all of the interconnected units, so you must ensure that there aren't significant differences in the ground potential of these units, or circuit damage could occur.

BasicX supports an independent COM port (defined as COM2), in addition to the Network port. Signal-level translation to RS-232 levels can be done with a MAX232. NetMedia chose instead to use a few transistors on the BasicX development board, stretching the RS-232 specification by transmitting data using 5-V logic levels instead of the ±3-V minimum levels specified by the RS-232 standard. It works fine this way.

Now you have a microcontroller with program memory, a network connection, and a serial port, which leaves 26 I/O lines free for other uses. Including the SPI interface, which can be used for other devices besides the EEPROM memory, you may have all

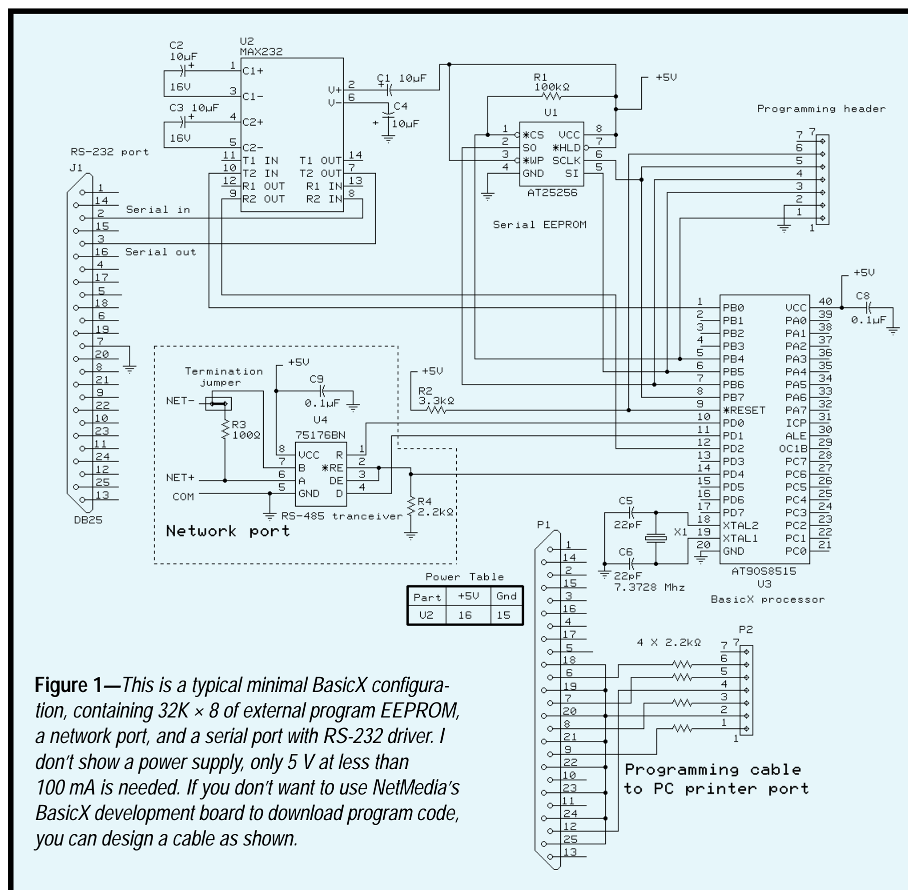


Figure 1—*This is a typical minimal BasicX configuration, containing 32K × 8 of external program EEPROM, a network port, and a serial port with RS-232 driver. I don't show a power supply, only 5 V at less than 100 mA is needed. If you don't want to use NetMedia's BasicX development board to download program code, you can design a cable as shown.*

of the I/O lines needed for your project. On the other hand, if it is an I/O-intensive project, you can switch the 'S8515 to the external SRAM mode, which provides a full 16-bit address and 8-bit databus that can be used for SRAM or extended I/O.

THE BASICX LANGUAGE

If you like what you've read so far, you'll be interested in the BasicX language. I prefer the user's program to be compiled into machine language and then stored in the 'S8515 flash memory, because compiled native code runs quicker than interpreted pseudocode programs. Furthermore, fetching pseudocode instructions from the SPI EEPROM is slower than accessing the 'S8515's internal flash memory. Photo 2 shows the serial EEPROM's SPI clock and data output lines. Note that each SPI operation takes about 5 µs. Addressing a byte of serial EEPROM memory requires sending a command byte followed by a 16-bit address; then the data is available to be shifted in. Although the first (non-contiguous) memory fetch requires 20 µs, consecutive memory locations can be read at 5 µs per byte. This is slower than the flash array's 735-ns access time.

Because many of the valuable features mentioned in the hardware section depend on this design decision, you must accept the speed penalty. Let's investigate some of the interesting features of the BasicX language.

NUMBERS AND MATH

I've always thought that if you're not using a lot of math in your program, you don't need a high-level language. Let's look at BasicX. BasicX provides the following standard types: Boolean, Byte, Integer, Long Integer, and Single Precision Floating Point.

In addition, BasicX also adds a few variants, available by using the new keyword in the dimension statement:

UnsignedInteger
UnsignedLong

Neither QuickBASIC nor Visual Basic support unsigned integers or unsigned long integers. Anyone who has tried to read or load 16-bit peripheral registers in these languages knows what a chore this can be and may welcome this feature. But, before counting on absolute rescue, read the manual about NetMedia's implementation of these unsigned types because there are a few limitations that will haunt you if you simply charge ahead with your coding.

The floating-point math routines are IEEE-standard, so the results should be the same as if obtained on your PC. Floating-point numbers are also stored in IEEE format, so arrays can be easily transferred to a host PC in a block move and properly interpreted by the host application. I was disappointed that BasicX doesn't provide a function to convert ASCII strings to floating-point or integer numbers (VAL function). Nor does it provide the STR function, which converts in the other direction. Because most human input/output is ASCII string-oriented, this is unfortunate.

Luckily, tucked away among the examples on the BasicX CD-ROM is the serial I/O folder containing the sample program `SerialPort.bas`, which includes some BASIC routines to handle the conversion of various numeric data types to string output. However, such routines tend to enlarge your own program.

NETWORKING

Networking was another feature that caught my eye in the BasicX advertisment. I haven't had an urge to create Internet appliances, but I often develop groups of modules used for data acquisition and control in the lab environment. In some cases, it is difficult to connect them all to the host PC's available ports.

Software packages, such as those available from Cimetrics Technology, provide RS-485 network drivers for popular microcontrollers. And, although sophisticated networking protocols like TCP/IP have their place, they are too complex for small microcontroller applications. The idea that all commands, regardless of how simple, have to be wrapped in a packet that could be hundreds of bytes seems inefficient for my purposes, even when coupled with a high-speed Ethernet link.

For the application I had in mind, I decided to design a network controller using a BasicX circuit connected to a PC's COM port. All other devices are interconnected using the BasicX high-speed RS-485 network interface. An offshoot of this network controller is the BasicX NetSnoop project, which is the subject of Part 2 of this series. This software provides a background debug monitor for BasicX systems.

BasicX networking differs from conventional network protocols. In a conventional network, the master sends an address-stamped message packet out on the network. All other devices on the network must monitor the packets, and the addressed node must recognize messages addressed to it. At this point, the addressed node must parse the incoming message into a command, possibly with one or more parameters. If the intent of that command is to illicit some response from the addressed node, it must also be able to take control of the network and return that value in a message packet of its own. It is a difficult task, given this protocol, to convert standalone firmware devices to work as networked devices.

BasicX bases its networking protocol instead on the concept of shared variables. In other words, rather than sending out a message that the receiving node must parse and act upon, the user's program issues a command to write a value to a particular variable at a particular network node address. Depending on the variable involved, this could be as simple as a single byte value or as complex as a whole array of data. Retrieving data from another node is also simply a matter of reading the value of a variable at a particular node address. Expanding on this concept, BasicX also supports the concepts of groupcasting and broadcasting, in which the variable-write functions can be targeted to like-numbered node groups or to the whole network, respectively.

Adding network functionality starts with adding the single line:

```
OpenNetwork(BA,GA)
```

where BA is the board address assigned to this particular BasicX module and GA is the group address assigned to this module. A number must be assigned, but groupcasting does not necessarily need to be used.

After this initialization, communication between modules is done using the GetNetwork and PutNetwork commands. Both commands require you to specify the board address of the intended node, as well as the name of the variable on the node that is to be read or written.

The other parameter needed is the value(s) to read and write to that variable. To simplify error handling, BasicX performs the requested data transfer until it is either successful or a timeout period has elapsed. The success or failure status of the operation is returned to the user in a result byte, which the program can employ to perform handling of errors or recovery routines.

The convenience of being able to refer to a remote node's variables by name is not something that comes without effort. It is the programmer's responsibility to include a copy of the MAP file(s) associated with the program code that is running on any node to which one intends to transfer data.

The MAP file is a text file (with an .MPX extension) that is automatically generated by the compiler, in addition to the program code file it generates for downloading to the BasicX chip. To do this, I copied the section of the map file containing the references to the variables of interest to the clipboard. Then, I pasted this section into the beginning of my BASIC program.

When you're declaring variables in a program, it helps to group network-related variables at the beginning of the list. Then, only that part of the map file has to be clipped and pasted into the BASIC code of the other modules that access those variables.

Incidentally, assembly language or C programmers who are used to the INCLUDE directive for handling such tasks, will find this procedure neither intuitive nor efficient. However, in my experience, BASIC source code files are the only files that appear to be able to be grouped together as part of a project. But, I have not received response from NetMedia to various questions, so I can only report my experiences gleaned from the documentation available in the package itself and from information on the company's web site.

The bottom line is that this network concept makes it easier to develop and debug a device as a freestanding unit and convert it easily to a networked device. And, I've saved the best part for last—the speed of this network is 460,800 bps. Although it is slow compared to a 10BaseT Ethernet link at 10 Mbps, the protocol is lean, resulting in respectable data transfer rates.

MULTITASKING

I must confess that in all of my years of programming, I have seldom written programs that could be described as multitasking. I often make full use of both internally- and exter-

nally-generated interrupts from timers, ADCs, and so forth, so it can be argued that, in this case, the processor is doing many things at once. Although many of my programs include code for a real-time clock, which often performs scheduled functions, I've never expanded on this to provide full-fledged task-switching and scheduling functionality.

BasicX provides this multitasking capability, the basis of which is the RTC that operates in the background during program execution. Its intrinsic timing resolution (tick) is 1.95 ms. Program tasks are given 1.95-ms slices of CPU time, with each task getting its turn. This task-switching rate is fast enough to be useful for many real-time operations, particularly those involving human I/O, we move slowly by computer standards. Tasks that perform critical operations can block task-switching, if necessary, to complete a particular job (the `LockTask` procedure). Conversely, lazy tasks, or those that are generally idle while waiting for infrequent I/O activity, can yield their slice to any other task that has work to perform.

Using multitasking in a program is not difficult for modest applications. Basically, a procedure can be written in much the same way as if it were called from a main program. When the program is restructured for multitasking, it calls this task using the `calltask` syntax, instead of the `call` syntax. Tasks that contain loops awaiting infrequent input events must be modified to yield their time-slice if no activity is pending. High priority tasks, or those that perform operations that cannot be interrupted even for a few milliseconds, must call the `LockTask` procedure at the beginning of the critical part of the code, and then the `UnlockTask` procedure when finished.

In any multitasking environment, the possibility exists that numerous tasks may reference common variables. The procedure for modifying variables is not inherently indivisible in a microcontroller. That is, one task could be part way through modifying a variable when its time-slice ends and another begins. The results could be unpredictable, depending on whether another task refers to or changes that variable. For this reason, a multitasking executive must introduce the concept of the semaphore, which is a way to limit access to a variable by any task other than one that started a modification operation on that variable. BasicX supports the semaphore concept.

The only obvious fly in the ointment with BasicX multitasking involves the allocation of stack space for the tasks. With or without multitasking, BasicX uses an implicit stack for its main program that is automatically allocated, meaning it is not a user declaration. Multitasking requires that a separate stack space be allotted by the user for each task.

There are no firm rules dictating how to calculate the necessary stack space, but if you underestimate it, your program will crash. Unless you are using external RAM, you only have about 250 bytes of internal RAM available for your variables, I/O queues, and stack space. Clearly, you can't pop in a high number here for safety's sake! Rest assured, though, the documentation does provide some hints along the way.

NEAT TRICKS

In addition to the normal complement of BASIC functions, NetMedia threw in smart functions and procedures to take advantage of the 'S8515 hardware features. I'll briefly outline a few that I found interesting.

`DACpin` is a procedure that provides a rapid set of precisely timed pulses to simulate an analog output voltage on any available I/O pin. Its resolution is about 8 bits and has to be called periodically to refresh the voltage at the pin.

By using an external variable resistor and fixed capacitor connected to a free input pin, the `RCTime` procedure returns a value proportional to the value of the resistor. It accomplishes this by shorting out the capacitor and measuring the time it takes for the capacitor to charge to the switching threshold voltage of the pin. This can be used for modest resolution measurements, such as reading resistive sensors or the value of calibration or front-panel user-adjustable pots.

The `InputCapture` procedure allows you to capture the transition times on the InputCapture pin of the processor. The desired number of transitions and the sense of the leading edge are user-defined in the parameter list of the procedure call. When called, an array is filled with these transition time values. However, shortcomings include the fact that this procedure suspends all other operations during its execution, hanging up the program if the required number of input transitions doesn't occur.

Complementary to the above procedure, `OutputCapture` sends out a user-programmable pulse train to the dedicated OutputCapture pin on the processor. The resolution is 135.6 ns, and the pulse widths are specified as unsigned integers. The phase of the pulse train is user-specified, and the complexity is limited only by the size of the array, which is prefilled with the individual pulse width values. As with `InputCapture`, this procedure completely ties up the processor during its execution. And it uses the 'S8515 Timer1, so it will interfere with the operation of COM2, which also shares this timer. If you use this function and also need to use the COM2 port (non-concurrently), reopen the COM2 port each time you use either of these capture procedures.

The `SPIcmd` procedure lets you access the processor's SPI port, allowing you to connect SPI peripherals to the processor. The parameter list of the associated `OpenSPI` procedure allows you to interface to many different SPI devices by setting clock rate, polarity, and phase. Refer to Photo 2 where the SPI bus is busy doing EEPROM memory fetches. Don't expect to be able to connect devices with high data-transfer requirements to this bus without a degradation of program execution speed.

PICTURE WORTH 1000 WORDS

Although NetMedia compares BasicX to Microsoft Visual Basic in its advertisements, this comparison is stretching it. BasicX is a beneficial implementation for a microcontroller.

However, Visual Basic is an event-driven language based on the Windows GUI, which draws heavily on many assorted graphical objects. BasicX has no intrinsic support for graphics. This is logical because the hardware has neither inherent graphics input nor output devices. Although BasicX syntax resembles Visual Basic, it more closely resembles QuickBASIC, which is not based on a GUI (although it did support graphics). Perhaps NetMedia thought many of its younger customers have not heard of QuickBASIC, or DOS for that matter!

THE ATMEL AVR PROCESSOR

One significant reason why programmers like high-level languages is because they insulate from the nitty-gritty details of the processor. However, for low chip-count embedded circuits, the processor design is important. As discussed earlier, because you can expect a considerable performance penalty going from native code to a pseudocode interpreter, it becomes necessary that the processor be both fast and efficient in terms of an instruction set. I'll briefly outline the architecture of the Atmel AT90S8515 chip used in the BasicX system, and I think you'll agree that it fulfills both of these criteria nicely. The Atmel 90S8515 microcontroller's datasheets, in PDF format, are included on the BasicX CD-ROM. You can also download them directly from Atmel.

The 'S8515 is part of Atmel's AVR family. It labels the AVR family as having RISC architecture, with 118 instructions and nine different addressing modes. If this is a reduced instruction set, I wonder what I should call the PIC chips I've been using lately, with their limited 37 instructions and few addressing modes. I assume that the RISC label on the AVR is justified because the instructions execute in one cycle. Also, unlike other processors that divide the clock input frequency by a factor between two and five, the 'S8515 runs at the full speed of the clock crystal, which ranges as high as 8 MHz. Coupled with its two-stage instruction pipeline, this yields an 8-MIPS rating. The 'S8515 processor has 32 general-purpose registers, directly addressable in one cycle, and three 16-bit address pointers for indirect data addressing. High-level languages are particularly well served by the numerous address pointers.

Program memory is flash EEPROM-organized as 4 K of 16-bit words to match the 16- or 32-bit instruction width. In the BasicX system, this memory space is pre-programmed by NetMedia with the BASIC pseudocode interpreter, so this leaves little more to say. Although this memory space is flash PROM and therefore easily erased, I assume NetMedia has used the Write-Lock feature of this chip to keep its program intact.

The 'S8515 contains 512 bytes of SRAM, with 250 bytes available for the user's BASIC variables, queues, and so on. The rest is reserved for the interpreter, but its needs vary. I've had programs crash with 190 bytes of RAM used for variables, and no compiler error messages were generated. This is fine for many applications, but if more SRAM is required, it can be accommodated using the external SRAM mode of the processor. By setting bit number seven (the external SRAM mode bit) in the MCUCR register, 19 general-purpose I/O lines of the 'S8515 switch functions to provide a 16-bit (multiplexed) address bus, an 8-bit data bus, plus the usual –RD and –WR lines. This allows for direct addressing of 64 KB of RAM, minus a small area that is mapped to the register file and the I/O registers. NetMedia sells a "RAM-sandwich" module, which uses this feature (with bank-switching) to provide 128 KB of SRAM. I have not tried this module but have used the external SRAM mode to successfully provide an address/data bus for external I/O chips.

Rounding out the onboard memory is 512 bytes of data EEPROM. In BasicX, this area is used for persistent variables (i.e., those that do not lose their value when the power is switched off). Like all EEPROM arrays, this memory can endure a finite number of erase/write cycles. (In the

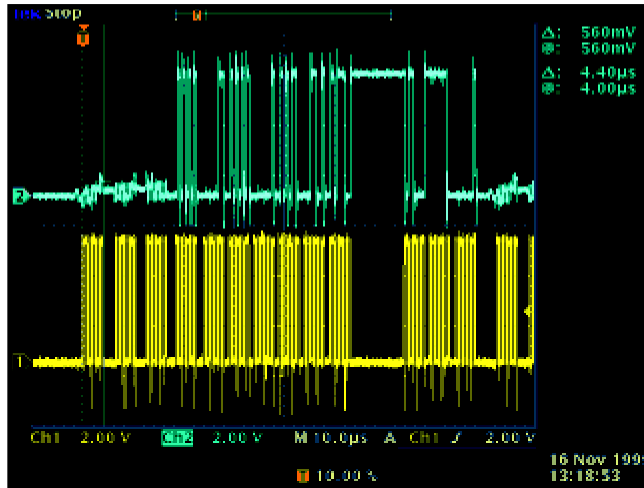


Photo 2—*The SPI bus on a BasicX system handles the program EEPROM I/O, and is an extremely busy highway! This 'scope image is typical during active program execution. The upper trace is the EEPROM data out line, and the lower one is the SPI clock. The first three clock bursts are for command/address data being sent to the EEPROM, therefore no data is seen on the EEPROM data out line during that interval.*

case of the 'S8515, the number is at least 100,000.) Common to all EEPROM cells, a write cycle takes approximately 4 ms to complete. Although this timing is handled transparently by the BasicX interpreter, it may have to be considered in some user applications. With these two limitations in mind, persistent variables are best left for things like configuration data that seldom changes but must survive power shutdown or short power interruptions.

Like most modern microcontrollers, the 'S8515 has a variety of built-in timer/counter modules. There is one 16-bit timer/counter and one 8-bit timer/counter, each with its own prescaler. The 16-bit timer/counter has an input capture register for monitoring the timing of signal changes on its InputCapture pin. Also, there are two compare registers that generate output pulses of accurate widths or implement a dual PWM feature. A notable feature of the PWMs is that they are glitch-free and phase-correct, which is important in some motor-control applications. To date, I haven't seen this level of sophistication in other microcontroller chips.

Luckily, both the InputCapture and pulse-generating features of this timer/counter are directly supported by BasicX commands, because any features involving accurate timing are of limited use unless directly supported by the language. Although the dual PWM functions are not intrinsically part of the BasicX language, you may refer to the References section at the end of the article for an application note describing some BasicX code that will implement this feature as a background task.

The 8-bit timer/counter is used by the BasicX operating system to implement an RTC with resolution of 512 ticks per second. The RTC also acts as the task-switcher for BasicX multitasking and handles the sleep procedure, which provides delays in 1.95-ms increments.

Asynchronous data communication is well supported by the 'S8515, with a full-featured UART port (referred to as COM1 in BasicX). This UART block contains its own data transfer-rate generator, so you don't have to waste a timer/counter for this purpose, as with other micros. The UART has 9-bit capability, which is used by the BasicX networking feature. When used for networking, BasicX uses the UART's highest speed of 460,800 bps.

BasicX also supports a second communications port, COM2. Don't look for this UART port in the 'S8515 datasheets. Instead, BasicX implements a bit-banged or software UART for this port, topping out at 19,200 bps, which is respectable for a software UART running in parallel with the user's code. Both COM ports use individual circular buffers for input and output, the length of which is user-programmable. The BasicX example programs use the 19,200-bps rate, and so far, I haven't experienced any data loss.

The last major block in the 'S8515 is the SPI port. Like most SPI blocks, it allows programming of clock rate, polarity, and phase. Additionally, it

allows you to choose the data order (i.e., whether the LSB or MSB of the data word is sent first). On several occasions I've been forced to use bit-reversal routines to accommodate specific peripheral ICs (my article in *Circuit Cellar* 95, "Digital Attenuators" comes to mind).

In the BasicX system, the user program is stored in an external serial EEPROM (an Atmel AT25256) connected to the SPI bus of the 'S8515 microcontroller. However, the SPI bus can also be used with other peripherals by providing a separate *Chip Select line for each one, using spare I/O lines on the microcontroller. In most cases, the SPI signal format required by these peripherals differs from that used by the EEPROM memory device. Earlier I mentioned the `SPIcmd` procedure which allows SPI transfers into and out of the 'S8515 with user-selected SPI parameters. If used properly, this doesn't disrupt the regular program memory fetches from the serial EEPROM, which would crash the program.

My last observation about the 'S8515 involves its power requirements. At clock frequencies less than 4 MHz, the chip will operate down to 2.7 V, making it usable in battery-powered devices. It works well in the 4.0–6.0-V range over the full-speed range, making it a good choice for circuits running from four 1.5-V cells.

WHAT'S NEXT?

In the second part of the article, I'll explain BasicX software. I'll also discuss a PC-hosted background debug monitor that I wrote to troubleshoot BasicX projects.

While you're waiting for Part 2, check out the Delphi BasicX forum for more BasicX information. It's been interesting uncovering yet another useful microcontroller. ◼

Brian Millier is an instrumentation engineer in Dalhousie University's chemistry department, Halifax, NS, Canada. He also runs Computer Interface Consultants. You may reach him at brian.millier@dal.ca.

SOFTWARE

The Visual Basic host program, as well as the BasicX firware, will be available to download free from the *Circuit Cellar* web site and Brian's site at bmillier.chem.da.ca.

SOURCES

AVR microcontroller
Atmel
(408) 441-0311
Fax: (408) 436-4309
www.atmel.com

BasicX
NetMedia, Inc.
(520) 544-4567
Fax: (520) 544-0800
www.basicX.com

REFERENCES

BasicX Software Updates, www.basicX.com/transfer

BasicX User Forum on Delphi, www.delphi.com/basicx/start

Atmel, www.atmel.com/atmel/products/prod23.htm

FEATURE ARTICLE

George Novacek

Right on Time

Quartz Crystals and Oscillators

In the pre-microprocessor era, quartz crystals were something most designers had only heard about, and most had heard very little at that. As George points out, even today many people still don't understand exactly what makes crystals tick.

Before the advent of the microprocessor, a quartz crystal was a mysterious component that was associated with ham radio amateurs, military personnel, and U.S. Post Office communications employees. Few people understood how it worked, and fewer knew how to design a circuit with it. Generations of electronics engineers never saw one during their careers. Color TV and the microprocessor changed that. The 3.579545-MHz color burst crystal, now manufactured in volume for consumer goods, lowered the price. Then, a microprocessor designer achieved accurate timing with a crystal controlled clock, and the rest is history.

But, one thing did not change. Although crystals keep precise clocks ticking inside nearly every electronic device today, few people understand what makes them tick. That is because manufacturers make life easy for engineers by producing ready-made, monolithic oscillators. To make a stable clock, buy a chip and plug in the crystal. Is there more?

Engineers will benefit from understanding how crystals work so that they can produce robust designs with optimum performance under adverse environmental conditions. In this article, I'll discuss the crystal's principle of operation first, then consider applications and oscillator design.

FROM THE BEGINNING

The reason crystals are used to produce stable frequencies is their inherent accuracy and stability. Oscillators need timing elements, which can be resistor-capacitor (RC), inductor-capacitor (LC), or electro-mechanical. Crystals, ceramic resonators, and tuning forks are electromechanical resonators. The best you can achieve with an RC oscillator is 0.1% accuracy and stability, an LC oscillator will do approximately 0.01% better. But, with a well-designed crystal oscillator, a few parts-per-million (ppm) accuracy is within reach.

Now, I'll state a few points about ceramic resonators and tuning forks, and then move on. Ceramic resonators differ from crystals in that they are manufactured from polycrystalline ceramic; quartz crystals are monocrystalline quartz. Ceramic resonators achieve approximately two orders of magnitude less accuracy and stability than crystals but are easier to manufacture and cost less. In applications where price is crucial and timing accuracy is not, they are acceptable. Principles that apply to the operation and application of quartz crystals are equally applicable to ceramic resonators. In fact, in many circuits, a ceramic resonator can substitute for the crystal. However, the price you pay is decreased performance and possibly a minor adjustment to the component values. Quartz crystals are practical at frequencies from 100 kHz to 40 MHz.

Previously, tuning forks, magnetically and piezoelectrically driven, were preferred for frequencies less

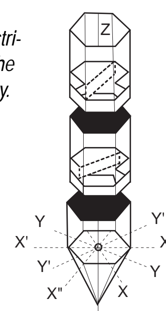


Figure 1—*Take a look at the detail cut and the optical, electrical, and mechanical axes in the typical quartz crystal geometry.*

than 100 kHz. Today, because microelectronics are inexpensive, it is practical to use crystal oscillators with signal processing instead. When an application requires it, divide the frequency to whatever is needed. When a waveform is required, it can be simply and inexpensively synthesized.

For frequencies higher than 40 MHz, crystal wafers become too thin and less practical to manufacture. Because crystals can oscillate on odd harmonics (third, fifth, and seventh), overtone oscillators and phase locked loop (PLL) are used to achieve high frequency oscillations. Surface acoustic wave (SAW) resonators are gaining popularity for frequencies greater than 100 MHz, and inverted MESA crystals run at several hundred megahertz.

With an effective L/C ratio, crystal resonators have high Q (quality factor), which determines their narrow signal bandwidth. The higher the Q, the narrower the bandwidth. Achieving Q = 150 at 1 MHz with a LC circuit is excellent; a crystal can provide 250,000. This means you can build a narrow band-pass filter or a stable, clean, low-distortion oscillator.

In its natural form, a quartz crystal has a hexagonal cross-section (see Figure 1) that's defined by three axes (*X*, *Y*, and *Z*). *Z* stands for an optical axis, which goes through the apexes of the crystal. The *X* and *Y* axes are perpendicular to each other and exist in three sets that are perpendicular to the sides of the hexagon: *X*, *X′*, *X″* and *Y*, *Y′*, *Y″*. The *X* axes are electrical, and *Y* axes are referred to as mechanical. The quartz crystal's piezoelectric properties mean that when you apply a voltage to an electrical axis, for example *X′*, a mechanical distortion will result in the corresponding mechanical axis *Y′*.

Conversely, if you stress the mechanical axis, a voltage will develop across the corresponding electrical axis. In a crystal resonator, electrical terminals are fixed to a small plate cut out of the original crystal (called a cut). An AC voltage applied to these terminals causes the cut to vibrate.

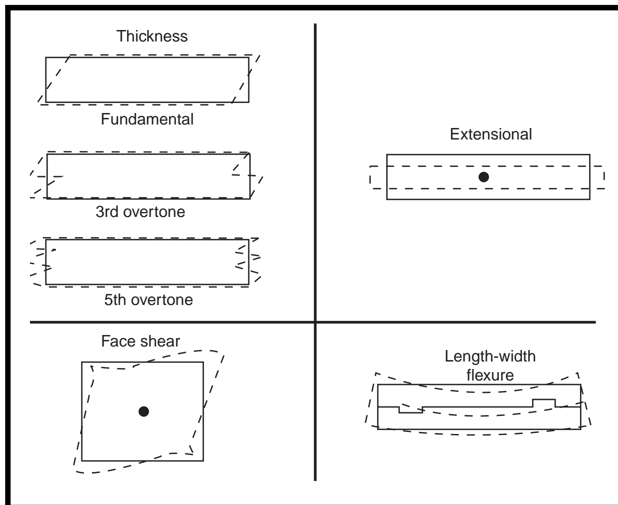


Figure 2—*Here are four examples of quartz crystal cuts' oscillation modes.*

When the vibrations occur at the cut's resonant frequency, their amplitude grows to a magnitude that is determined by the quality factor Q. The resonant frequency of the cut is determined by the cut's mechanical characteristics (such as geometry, size, and thickness) and the way it was cut out of the original crystal. Invisible to you, the cuts oscillate in different modes, depending on the design. This characteristic and some modes of vibration are illustrated in Figure 2.

The extensional oscillations are typical of the most commonly used cut, AT, and the face shear would be seen in cuts like CT or GT. There are infinite combinations of crystal cuts with varying characteristics. For convenience, the industry settled with a number of commonly used cuts. Figure 3 shows them. Today's applications use the AT cut most often.

FREQUENCY STABILITY

The primary factors that contribute to the crystal oscillator's frequency stability are:

- temperature at ±50 ppm, from –55°C to 125°C
- power supply at ±2 ppm for 5 V, ±10%
- load at ±1 ppm for 20% load change
- time is responsible for short-term ±0.01 ppm per second
- ages ±15 to 20 ppm per year

If the crystal is properly packaged and operates with well-designed electronics, temperature is the most significant influence on its frequency stability. The type of the cut determines this. As you see in Figure 3, the AT cut is made at 35° to the *Z* axis. Varying the 35° angle by 14 angular minutes (+10′, –4′) affects the frequency versus temperature performance (see Figure 4). Today's manufacturing process is accurate and well controlled. The best in class (BIC) manufacturers determine the optimum angle for the desired operating temperature range and manufacture the crystals accordingly. For example, because 2′ offset will result in the minimum temperature drift for 0°C to 70°C, using crystal outside these limits will deliver less than optimal performance.

For the military operating range (–55°C to 105°C) you can achieve ±50-ppm frequency stability. If this isn't good enough, use a crystal with an integral oven that keeps it at a constant temperature. Such compensated crystals are available in varieties that depend on the principle of compensation control called TCXO, OCXO, MCXO, and so on. They deliver $\pm 3 \times 10^{-8}$ frequency stability and aging autocalibration. Don't forget the common electronic wristwatch, which is often clocked by an inexpensive ceramic resonator. How can it be so accurate? Its success relies on the human body—an excellent temperature controller. Wearing the watch maintains its internal operating temperature at an almost constant 37°C.

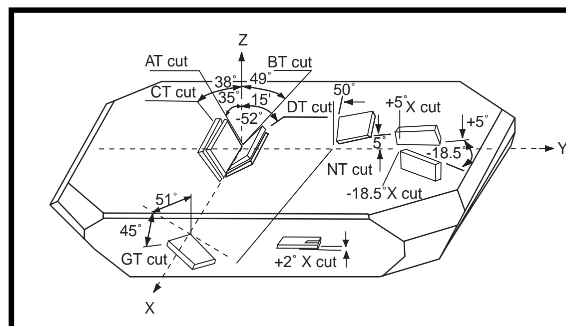


Figure 3—*This is how crystal cuts are created. Notice the most common cut, AT, which varies from BT, CT, and DT by one angle to the Z axis.*

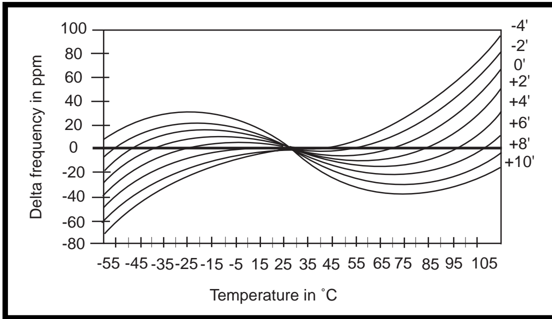


Figure 4—*This is a relative AT cut angle chart.*

Another cause of frequency instability is aging, which is caused by gradual relief of the mechanical strain on the crystal. If you purchase preaged crystals, expect about 15 ppm per year. This means a 10-MHz oscillator may drift 150 Hz during one year.

ELECTRICAL CHARACTERISTICS

Electrically, the quartz crystal can be modeled as a resistance inductance capacitance (RLC) network (see Figure 5a). R_m, L_m, and C_m, also called motion characteristics, represent the cut's mechanical characteristics. Parallel capacitance (C_o) is the result of the mounting hardware and the electrical terminals attached to the cut. In a parallel oscillator circuit, external capacitances are added to it.

The theory of electrical circuits states that whenever inductive and capacitive impedances are equal, the LC network resonates. The crystal exhibits serial resonance at frequency

$$f_r = \frac{1}{2\pi \cdot \sqrt{L_m C_m}}$$

The crystal's effective inductance (L_m) is often in henrys (H), while the capacitance (C_m) is in low, even fractions of a picofarad (pF). This acounts for the high Q. Resistance (R_m) represents the loss of the resonator, which, in an oscillator, must be compensated by the amplifier gain to sustain oscillations.

The topology of the circuit in Figure 5a indicates that two resonances, one series determined by the motion components and one parallel due to C_0, will exist. The parallel resonance is approximately 1% higher than the series resonance (see Figure 5b). The parallel resonance can be pulled slightly by external components.

If you determine the stray capacitances exactly, the new resonant frequency would be:

$$f_a = f_s\left(1 + \frac{C_m}{2C_0}\right)$$

This property trims crystal oscillators operating in parallel resonance mode. It is the basis for the voltage-controlled crystal oscillator (VCXO) where frequency deviation between ±10 and ±100 ppm can be achieved.

Although series resonance affords the best frequency stability, parallel resonant oscillators (Pierce's topology) are used more because they're simple, especially in microprocessor circuits. As a result, series mode crystals are difficult to find. Structurally they are the same as parallel crystals. They differ only in testing and the resonant frequency marking.

A parallel crystal can be used in a series mode oscillator and vice versa, but the parallel crystal in the series oscillator will run at a lower frequency than marked.

THE OSCILLATOR

Now that you understand the principles behind the ubiquitous quartz crystal, you can make a stable oscillator with it. For an amplifier to oscillate, it must satisfy two conditions at the frequency of oscillation.

First, it must have a positive feedback, that is 360° (0°) phase between the input and the output. Secondly, its open-loop gain must be greater than one at the oscillation frequency. This is known as the Barkhausen criterion. The resonator suppresses the gain outside the desired frequency and provides a needed phase shift.

Figures 6a and 6b depict simplified diagrams of two of the most popular oscillator configurations. They are shown with inverter circuits for two reasons. Their operation is easier to understand than if they were shown with discrete transistors, and this is how the majority of today's oscillators are built.

For completeness, Figure 7 shows several discrete oscillator designs, with both bipolar transistors and unipolar transistors. Today, these oscillators seem to be limited to specialized applications in which a monolithic design won't suffice, or in toys and consumer goods where every penny counts.

Consider the series resonant oscillator exhibited in Figure 6a. The two inverters perform the 360° phase shift, while the crystal in the feedback exhibits minimum attenuation (its impedance is resistive, equal to R_m) at the resonant frequency. The first inverter is biased by R1 to operate in the linear region. The second inverter drives the crystal with a square wave. Because of its high Q of 10^5 and inverter gain drop-off at higher frequencies, the harmonics contained in the square wave are suppressed, and a pure sinewave can be observed at the first inverter input.

Using Pierce's topology in Figure 6b, the parallel resonant oscillator is a common design today and is used as the basis for internal microprocessor clocks. Unlike its series-resonant brother, it needs only one inverter, which provides 180° phase shift. Another 90° shift comes from

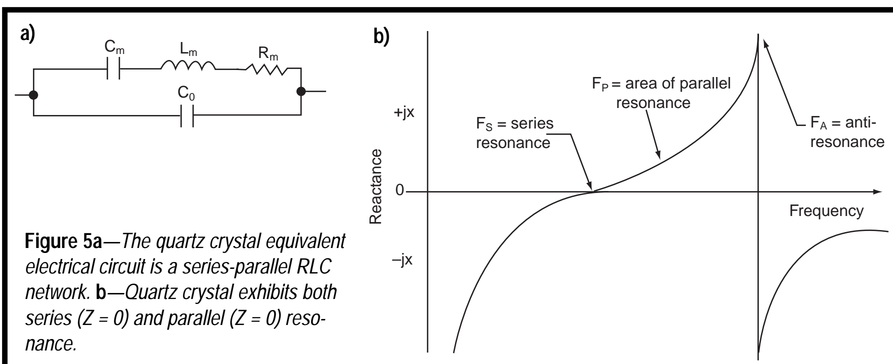


Figure 5a—*The quartz crystal equivalent electrical circuit is a series-parallel RLC network.* **b**—*Quartz crystal exhibits both series (Z = 0) and parallel (Z = 0) resonance.*

the R2C2 element. The crystal, vibrating at its series resonance, appears to the rest of the circuit as resistor R_m, which, together with C1, adds a 90° shift for a total of 360°. The inverter is biased into its linear region by R1. And, if it has sufficient gain at the crystal resonant frequency, the Barkhausen criteria are satisfied and oscillations ensue.

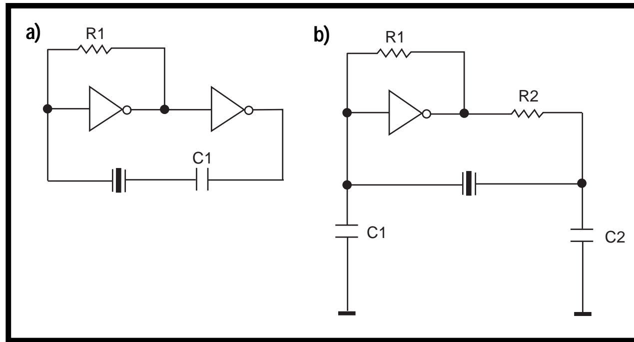


Figure 6—*Here is an example of a series resonant oscillator **(a)** and a parallel resonant oscillator **(b)**.*

That works theoretically. The reality is not as ideal. First, especially at higher frequencies, the inverter will have internal delays, and its phase shift will be greater than 180°. It's not uncommon to reach 185°. Secondly, R2C2 will more realistically generate a phase shift of less than 90°, approximately 73°. To oscillate, the frequency will have to shift above series resonance, where the crystal impedance changes from purely resistive (R_m) to slightly inductive until the total phase shift hits 360°. This is why it is possible to pull the crystal frequency by changing the external capacitance. Similarly, the series resonant circuit often needs a capacitor in series with the crystal (C1) for phase compensation.

OSCILLATOR DESIGN

How do you put together everything you have learned and build an oscillator? Making an amplifier oscillate is a piece of cake, but usually an undesirable one. Making it oscillate in a stable, predictable way is a different story. The deceptively simple circuit with six components has many glitches. Few components are sufficiently characterized for precise calculations, and the components' tolerances make the calculations just good enough to make it into the ballpark. Therefore, unless you plan to make crystal oscillator design your career, I recommend that you become an educated buyer.

Most microelectronic circuits that require a precise clock have an onboard internal clock oscillator. Usually it is a Pierce circuit, and the external component values (C1 and C2, approximately 30 pF each) are specified. But, this does not prevent problems, especially when you need a volume-manufacturing consistency. The example described in reference [1] shows that the manufacturing tolerance of the crystal's loss resistance R_m combined with the internal CMOS

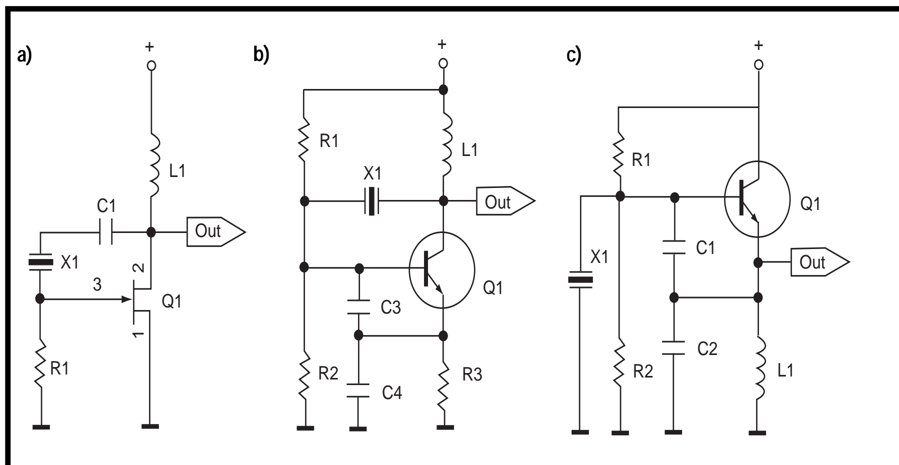


Figure 7—*The Pierce oscillator **(a)** is shown here, **(b)*** unplug with *a Colpitts oscillator, **(c)** and a Clapp oscillator.*

amplifier can cause the open-loop gain to drop below one, meaning the clock occasionally stops.

In marginal sysems, this may be difficult to detect because it manifests itself by only an occasional data corruption. The clock problems at startup can be further aggravated by temperature extremes. Here, author S. Dickey devised a clever solution with an inexpensive LC phase shift network, rather than a conventional, costly crystal solution. [1]

Another problem is a large gain, which may cause the crystal to resonate at an overtone frequency or, even worse, dissipate too much heat and crack. The only element within the crystal dissipating energy is R_m.

To calculate the crystal dissipation, you need to know the current passing through the crystal. The voltage across the crystal contains reactive components, so it is unusable for the calculation. Measure the current with a current probe, or insert a small resistor between the driver output and the crystal (in series with R2 in Figure 6b, which usually is an integral part of the driver circuit and is inaccessible) and measure voltage across it.

Crystals are specified for maximum power dissipation, usually in milliwatts. Excessive power can be reduced with a resistor in series with R2. Because this resistor also reduces the open-loop gain, be careful not to create the startup or stability problems described in the previous paragraph.

I'll note one more potential problem that I discovered the hard way many years ago. Make sure there is no constant DC voltage bias across the crystal. The quartz crystal does not conduct DC current, a fact that I verified with a VOM years ago.

So, I assumed that the few cents saved by eliminating a DC blocking capacitor from the oscillator, multiplied by thousands of dollars in the production run, would save a bundle. I was surprised when the oscillator exhibited unforeseen behavior that I could not explain. Eventually, I realized that the DC bias I allowed to exist across the crystal caused it to deform, in addition to the oscillator–driven vibrations. The result was strange and unacceptable. Although this is not a usual problem with today's monolithic oscillators, keep it in mind.

PRACTICAL APPLICATIONS

If you need an inexpensive crystal oscillator with no special requirements, use the circuit in Figure 6b. A CMOS inverter, such as 4007 or 4049 with R1 = 4.7 to 10 MΩ and C1 = C2 = 15 to 33 pF works well for most applications. CMOS inverters have high-output resistance, so R2 could be zero for a 5-V operation. But, a resistance may have to be added to limit the current in case of an overdrive at a higher supply voltage, or together with C2 to lower the critical frequency of the R2C2 low-pass filter if the oscillator runs at an overtone (harmonic) frequency. One additional inverter should be used as a buffer to isolate the load from the oscillator, thus improving stability.

If you want to build a microprocessor clock and the oscillator circuitry is onboard, follow the manufacturer's recommended design and component values. If you need an exceedingly stable clock for timing critical applications, you may have to build an external oscillator, so read on.

If you're determined to build your own, Motorola manufactures the crystal oscillator integrated circuit MC12061 that contains many bells and whistles, such as an AGC (automatic gain control). The specification sheet shows frequency drift of approximately 10 ppm through the military temperature range, that is from –55°C to 125°C (–67°F to 257°F), but this does not include the crystal. The IC can be configured to output sinewave, MECL, or TTL level signal. You still need a stable crystal, and be careful when laying out the PCB.

For an inexpensive oscillator with guaranteed performance, go to manufacturers like Q-Tech Corporation, Fox Electronics, and US Crystals to purchase a properly packaged and characterized, hermetically-sealed oscillator. With a prepackaged oscillator, your only tasks are providing power and keeping the load within the specifications. ■

George Novacek has 30 years of experience in circuit design and embedded controllers. He currently is the general manager of Messier-Dowty Electronics, a division of Messier-Dowty International, the world's largest manufacturer of landing-gear systems. You may reach him at gnovacek@nexicom.net

REFERENCES

[1] S. Dickey, "DSP IC's Clock Oscillator Uses Inexpensive Crystals," EDN, March 2, 1998.

[2] Fox electronics, "Quartz Crystal Theory of Operation," Fox Electronics, www.foxonline.com, 1999.

[3] "Military Specification Sheet, Crystal Unit, Quartz, CR64/U," U.S. Department of Defense, MIL-C-3098/42G 5, June, 1989.

[4] P. Horowitz, W. Hill, "The Art of Electronics," Cambridge University Press, NY, 1997.

[5] B. A. Smirenin, "Radiotechnicka Prirucka," SNTL, Prague, 1955.

[6] C. Turner, "Use of TMS320C5x Internal Oscillator With External Crystals or Ceramic Resonators," Texas Instruments SPRA054, October 1995.

SOURCES

Oscillators

Motorola
(800) 331-6456
(512) 328-2268
Fax: (512) 891-4465
www.motorola.com

Q-Tech Corporation
(310) 836-7900
Fax: (310) 836-2157
www.q-tech.com

Fox Electronics
(941) 693-0099
Fax: (941) 693-1554
www.foxonline.com

US Crystals
(800) 433-7140
Fax: (817) 923-0424
www.uscrystal.com

NOUVEAU PC

Edited by Harv Weiner

IDE DISK ADAPTER

The **DA104 IDE** disk adapter is a PC/104-compliant carrier board that provides a convenient means of integrating a 2.5″ IDE hard drive, M-Systems CompactFlash or M-Systems 1.3″, and 1.8″ IDE2000 Flash Disks to a PC/104 stack. A standard 40-pin IDE ribbon cable connects the drive to the host. Power can be supplied through the PC/104 headers or the two-position screw terminal.

For additional disk space, a second 2.5″ hard drive or IDE2000 Flash Disk can be mounted to the underside of the module. However, this configuration exceeds the PC/104 height specification.

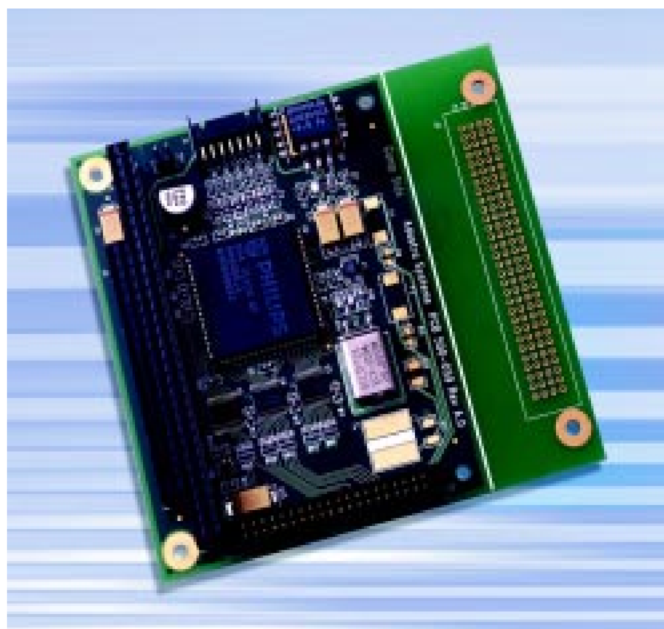
The DA104 sells for **$99**. The underside drive interface costs $30 (the IDE drive is not included).

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

DIGITAL VIDEO MODULE

The **DV-104** digital video module digitizes live video (NTSC, PAL, or SECAM) and outputs the digitized stream on a zoomed video bus. When combined with a zoomed video-capable single board computer (SBC) from Adastra's Venus SBC family, the system can display live video in a window on a SVGA monitor (CRT or flat panel) and capture video at 30 frames per second.

The DV-104 comes in three forms—PC/104, PC/104-Plus, and standalone. The standalone form is a compact 3.6″ × 2.8″. Its features include video digitization up to 640 × 480 pixels (NTSC), six composite or three S-video inputs or combinations, and 3.3-V digital video (zoomed video) output.



The DV-104/Venus combination is supported by a sophisticated software package. The support software defines an API that allows applications to display, manipulate, and capture live video in GUI and non-GUI environments. Versions of the software are available for Windows, QNX, and Linux, and support for other operating systems is planned. Sample applications demonstrate video windowing for QNX running Photon microGUI and Linux running X-Windows. The Windows version supports the latest version of DirectX, version 7.0, and an OCX control.

The DV-104 sells for **$95** in quantities of 100.

Adastra
(510) 732-6900
Fax: (510) 732-7655
www.adastra.com

Ingo Cyliax

Real-Time Executive for Multiprocessor Systems

Part 2: Running i386 RTEMS Applications

Having explained some of the background details of RTEMS last month, Ingo figured it would be a good idea to provide a practical application before people started setting up their own backyard missile command centers.

Last month, I introduced RTEMS, an open source real-time executive for 32-bit systems. I walked you through installing and building the host development environment.

One of the issues with setup is that it's based on the GNU C tool chain. This allows flexibility, but you must run it on a system that supports the tool chain. The best supportive environment for this is Linux. But, you can work with other Unix and Unix-like OSs. With Cygwin, a GNU environment for Windows, you can run it under Windows, with some effort.

Last month you built an environment that allows you to compile and link programs with the RTEMS kernel for the pc386 platform. Remember, pc386 is a board support package (BSP) for RTEMS that supports RTEMS to run on PC/AT-compatible systems.

The standard build constructs some example programs. This month, I'll cover how to run these, and build your own applications.

GRUB

The standard environment generates the libraries, drivers, and startup code for BSPs, and builds example programs. For the pc386 BSP these can be found in the directory `/opt/rtems/pc386/samples`.

All of the pathnames I refer to assume that you use `/opt/rtems` as the prefix for `configure--prefix=/opt/rtems`. This is also called the top level. You should have the `/opt/rtems/bins` directory in your search path because it contains the development tool chain components.

The sample programs are `base_sp.exe`, `minimum.exe`, `ticker.exe`, `hello.exe`, `paranoia.exe`, and `unlimited.exe`.

Don't be fooled by the `.exe` extension. They are not Windows nor DOS executables. They are 32-bit ELF executables that contain the whole run-time image for an application. This means the application's main code, RTEMS kernel, drivers, library modules, and startup code needed to prepare the board to run RTEMS are included in this file.

To proceed, a target is needed. A target is the system that will run the application code, as opposed to the host system, which is used to build the application. I've been using two different systems as a target. One system is my laptop. In this case, I reboot it to run RTEMS applications for quick tests. Another test system is a spare computer that serves as my embedded development test platform (see Photo 1). The procedure for getting RTEMS to run is the same—build a boot floppy for the target to boot from.

Photo 1—*My trusty embedded test-bed. It's a discarded P120 machine, parts of which are least six years old. I don't think it ever had a case, so it is simply bolted to a piece of wood along with the power supply.*

A real system might use a flash PROM as the boot device or even a BIOS extension ROM. Booting on a PC/AT platform is complex. There is a BIOS that initializes all of the hardware and chipsets that are present. It then looks for a BIOS extension ROM, and finally a boot device, which is a floppy, flash–memory disk, or hard disk. With a BIOS extension ROM, you can boot from nonstandard boot devices. For example, an extension ROM can be used to download a boot image from a server to run it.

Listing 1—*In order to prepare a boot floppy for RTEMS, you have to prepare two floppies. Floppy #1 is the Grub master floppy, and floppy #2 is the RTEMS boot floppy.*

```
#
# copy grub master to floppy #1
#
dd if=grub-boot-0.5.93.1.image of=/dev/fd0 bs=36b
     #
     # copy the stage1/stage2 file from flopp #1
mount /dev/fd0 /mnt
cp /mnt/boot/grub/stage{1,2} /tmp
umount /mnt
#
# change floppies and initialize a DOS file system on floppy #2
 mkdosfs /dev/fd0
 #
# mount the DOS floppy
 mount /dev/fd0 /mnt
 #
# copy stage1/stage2 on DOS floppy
 cp /tmp/stage{1,2} /mnt
#
# compress and copy the executables from the sample directory
foreach i ( *.exe )
gzip < $i > /mnt/$i:r.gz
end
#
# edit and add the file to the grubmen file on the floppy
# title= Hello World Test
# kernel= (fd0)/hello.gz
# ...
#
vi /mnt/grubmenu
#
# unmount floppy #2
#
umount /mnt
```

Listing 2—*Actually, this sample Grub menu file is more like a script file, but in this case, you only specify enough to implement a boot menu.*

```
title= Base SP
kernel= (fd0)/base_sp.gz
title= Hello World Test
kernel= (fd0)/hello.gz
title= Minimum
kernel= (fd0)/minimum.gz
title= Paranoia
kernel= (fd0)/paranoia.gz
 title= Ticker
kernel= (fd0)/ticker.gz
title= Unlimited
kernel= (fd0)/unlimited.gz
```

Listing 3—*Not much to a sample* `hello world` *program.* `Init()` *is the application initialization function. You would typically fire off tasks from here. In this case, you print a string.*

```
#include <bsp.h>
#include <stdio.h> rtems_task Init(rtems_task_argument ignored)
{
       printf("Hello World\n");
 }
 #define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
 #define CONFIGURE_INIT
 #include <confdefs.h>
```

If you want to boot a PC without the BIOS, the startup code would need to initialize the hardware in a way that the operating system or application expects to find it. RTEMS does have a BSP that allows build for a standalone boot PROM for an i386ex evaluation board. However, this is for a specific platform (the i386ex EVB from Intel), and expects the peripherals in an i386ex and on the EVB.

The best and easiest way to get an RTEMS application to run on a PC/AT compatible board is to use the BIOS to load the code from a boot device. By the way, if you are interested in seeing how to build a standalone boot PROM for a PC/AT board, drop me a note and I'll consider addressing this issue in a later article.

Now, how do you get a 32-bit ELF image loaded from a floppy so it will run? The answer—you have to use a boot loader. A boot loader is a piece of software that resides on the boot device (floppy, in this case) and bootstraps the application code into the memory of the system. The BIOS will load the first block of a bootable disk to find out what the geometry of the boot device is and how many more blocks to load from the boot loader. Figure 1 shows the general format of the master boot sector.

There are several boot loaders available. The ones that come with Windows or DOS are not too flexible (primarily for loading Windows or DOS). LILO is a Linux boot loader that could be used because it's freely available, includes source code, and works well overall. Most Linux systems use it as their boot loader. LILO is even capable of multibooting up to 16 different systems. I showed you how to use LILO in my Embedded RTLinux series a while back, so this time I'll use Grub, another widely available GNU-licensed boot loader.

Grub will boot almost anything from any file system type. Its use in booting RTEMS objects is documented in the pc386 BSP. It's a no-brainer for this application.

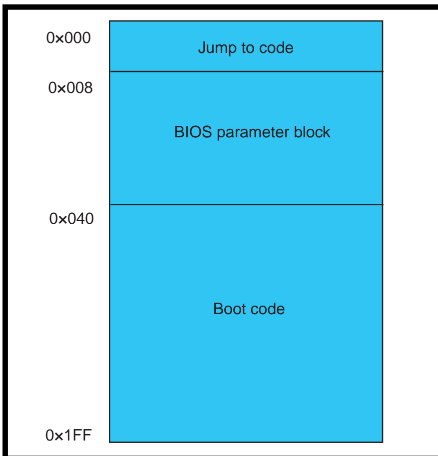


Figure 1—*Here is what the boot sector of a disk looks like. It contains the geometry for the disk, the primary partition table of the disk, and boot code.*

One feature that I will be using in Grub decompresses compressed boot images. This means I can compress my `.exe` samples file and fit them on a boot floppy. What do you need to use Grub? To start, two floppies are required. Next, the Grub masterboot floppy image (grub-boot-0.5.93.1.image, downloaded from prep.ai.mit.edu) is needed, and stage1 and stage2 files from the master boot file. Now, let's discuss the compressed executable that you want to boot and a config file that allows Grub to display a menu. And, that's the necessary list.

First, you have to build the two floppies. Listing 1 shows how to do this under Linux. There are instructions under Windows/DOS in the RTEMS BSP for pc386.

Floppy #1 is the Grub master floppy. Once you have generated this floppy, you can write-protect it because it will be used to initialize floppy #2. Floppy #2 is the RTEMS boot floppy. I compressed and copied the RTEMS images to it and built a small menu file (see Listing 2) that Grub will use to prompt me for the particular sample program to run. Also, you need a copy of the stage1 and stage2 Grub boot images in order to initialize the boot floppy.

After making the two floppies, you're ready to initialize the RTEMS boot floppy. When you boot the master floppy, a Grub installation menu is presented. You must select the entry that starts a command line interface, replace the master floppy with the RTEMS boot floppy, and issue the following command:

```
install=(fd0)stage1 (fd0)
 (fd0)stage2 0x8000
 (fd0)grubmenu
```

The command uses the stage1 file to construct the boot sector of the floppy and link to the stage2 boot file. Next, it tells the loader to use the `grubmenu` file to display after booting. After this is done, when you reboot the machine, you should see a screen resembling Photo 2. Then, select the RTEMS sample program you want to run. For instance, running the `hello world` example will leave you with a screen like Photo 3.

After each program, you will need to reboot the machine from the RTEMS boot floppy. Remember that each one of these programs is a complete RTEMS application image. It takes over the machine, does its thing, and upon completion, needs to be

rebooted. A real application unlikely finishes unless it encounters an error or the machine is rebooted on purpose.

BUILDING

Let's walk through building your program under RTEMS. First, you write a program; I followed the `hello world` example. The main part of the code is simple (see Listing 3). Define an `Init()` routine that prints a string on the console. To make sure you pick up the right pieces, include the files `bsp.h` and `confdefs.h`. The `#defines` at the end designate which parts of RTEMS you need for this module.

To compile, you need a `Makefile`. Listing 4, a generic `Makefile` that can build a variety of programs, works well. You want to build `test.exe` from the source file `test.c`.

One more thing is needed. This `Makefile` uses prototype `Makefiles` to form the RTEMS installation environment and figure out where to find the tools and libraries to include files. These prototype `Makefiles` are organized by BSPs, and in order to use the one for pc386, you need to set the environment variable:

```
RTEMS_MAKEFILE_PATH=/opt/
 rtems/pc386
```

Then, set the command (`make`) and out pops the `executable test.exe`, which is in the `o-optimize` directory. Then, copy it onto your RTEMS boot floppy and modify the Grub menu file. Delete the stage1 and stage2 files from this disk to make space.

NETWORKING

As stated, RTEMS has a TCP/IP network stack, a collection of routines that implement a protocol stack. It's referred to as a stack because the different layers of protocols stack up to implement a protocol system. Each protocol layer communicates with the layer above and below it and implements a virtual communication channel with the same layer in the protocol stack of a remote host.

There are several layers for Internet protocols. There is something called a physical layer that is closest to physical wire. If you are running Ethernet, this layer defines the electrical signaling characteristics of how to transmit frames over the wire. For example, Ethernet uses Manchester encoding.

The next layer up is the link layer. This layer defines how frames are constructed over the physical layer. For example, Ethernet uses a type HDLC framing with source and destination addresses and frame type fields. The link layer allows packets to be sent across the wire.

The Internet layer is the layer in the Internet protocol stack that allows host-to-host addressing and routing. It also implements a method in which packets that are larger than the link layer imposed frame length can be broken on the source host and then reassembled on the remote host. This disassembly and reassembly is transparent to the network layers above the Internet layer. Note the distinction between the Internet protocol stack and the Internet layer. The Internet layer is a generic term and can imple-

ment a different protocol for other protocol stacks (like Novel IPX), while the Internet Protocol (IP) is the protocol used at the Internet layer in the Internet Protocol stack.

Above the Internet layer is the presentation layer, which is the protocol layer that is presented to the application. In the Internet Protocol stack there are several different protocols—Unsequenced Datagram Protocol (UDP), Transmission Control Protocol (TCP), and Internet Control Message Protocol (ICMP). TCP and UDP are used by applications, and ICMP is used between hosts and routers.

TCP and UDP allow several different applications on hosts to communicate by assigning them port numbers. A connection is uniquely identified by the source port and host and the destination port and host. The endpoints are like sockets, and many TCP/IP protocol implementations call the file handle used by the applications to reference to these port/host pairs' sockets. TCP protocol allows stream-based communication between

Listing 4—*The `Makefile` is more complicated than the actual program. It is a general purpose Makefile that can be used to build more complex projects. Just change the value for EXEC and add whatever source modules you have to the CSRC (C source), CXXSRCS (C++ sources), and ASSRCS (assembler sources), and `make` figures out the rest.*

```
#
# Makefile
#
#
# RTEMS_MAKEFILE_PATH is typically set in an environment variable
#
EXEC=test.exe
PGM=${ARCH}/$(EXEC)
 # optional managers required
 MANAGERS=io
# C source names
CSRCS = test.c
COBJS_ = $(CSRCS:.c=.o)
COBJS = $(COBJS_:%=${ARCH}/%)
# C++ source names
CXXSRCS =
CXXOBJS_ = $(CXXSRCS:.cc=.o)
CXXOBJS = $(CXXOBJS_:%=${ARCH}/%)
 # AS source names
ASSRCS =
ASOBJS_ = $(ASSRCS:.s=.o)
ASOBJS = $(ASOBJS_:%=${ARCH}/%)
# Libraries
LIBS = -lrtemsall -lc
include $(RTEMS_MAKEFILE_PATH)/Makefile.inc
include $(RTEMS_CUSTOM)
include $(PROJECT_ROOT)/make/leaf.cfg
OBJS= $(COBJS) $(CXXOBJS) $(ASOBJS)
all:  ${ARCH} $(PGM)
$(PGM): $(OBJS)
       $(make-exe)
```

hosts. So, that means that bytes go in on one side and come out on a different side of a virtual pipe.

UDP communications are done with datagrams. Some pitfalls are that the datagrams may not arrive at the remote host, and they may be accidentally duplicated by routers along the way. TCP, on the other hand, is more robust because it operates transparently.

Bytes are not duplicated, and if packets get lost, they are retransmitted. Also, when the communication channel between two TCP endpoints ceases, the protocol will eventually time-out and signal the application that the connection does not exist anymore. The negative characteristic of TCP is that it has more overhead, both in the size of the packets sent over wire and the power required to process them.

Application layer protocols are the next layer up. These are not defined by most Internet protocol stack implementations. An application layer protocol in the Internet will use either TCP or UDP and implement a protocol that is suited for specific applications on top of these channels. For example, e-mail uses the simple mail transaction protocol (SMTP) to send mail between Internet hosts. Another application protocol is the post office protocol (POP), which your home PC uses to retrieve e-mail messages from a mail server at your Internet service provider (ISP) or office. Finally, the hypertext transaction protocol (HTTP) is used by your web browser to retrieve web pages from web servers.

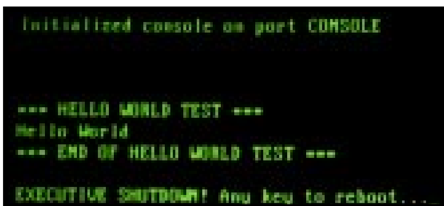


Photo 3—*Voilà! It works. After many hours of sweat and tears installing and building the RTEMS development environment, one of the sample programs works on my laptop and on my hardware test-bed.*

Photo 2—*The boot menu displays a line for each application image. Highlight the one you want and hit enter. Grub can also be programmed to have a timeout value, after which it will boot a default.*

The RTEMS protocol stack includes implementation of the Ethernet protocol, device drivers for several Ethernet cards, and the protocol stack up to the application layer. RTEMS also has PPP drivers. The protocol stack is based on the FreeBSD protocol stack, a freely available Unix operating system. This protocol stack is derived from the BSD protocol stack, which was used as the reference implementation.

RTEMS also includes an open-source web server implementation from GoAhead Software and an FTP server implementation. The web server and FTP allow file transfers to and from memory, so they operate truly as embedded web servers.

It's time to wrap up. Next month, I'll demonstrate how to bring up the web server and other Internet applications. ◾

Ingo Cyliax has written for Circuit Cellar *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. You may reach him at cyliax@derivation.com.*

EPC Applied PCs

Fred Eady

Picking Some ExacTicks

Keeping Precise Time

They say timing is everything, so Fred decided to take a look at the details of ExacTicks. If you need high-precision timing for delays, timers, or alarms, you might want to listen up. Of course, it all started with a blue box….

Microsoft is still keeping Windows NT 4.0 Embedded close to its chest. Not to worry, though, because I just got off the phone with Mark at Arcom Controls all the way in the U.K. It seems that Mark and Steve (his U.S. counterpart) were responsible for making another little blue box appear on my doorstep. Inside was an Arcom SBC-MediaGX. I've described this to you before, but for those of you who want to stick to this subject for now, here's the rundown.

WHAT'S IN THE BOX?

The SBC-MediaGX is composed of a National/Cyrix MediaGX 233-MHz MMX-enhanced processor and the National/Cyrix CX5530 I/O Companion chipset. There's 16 KB of L1 write-back cache supported by Award Software PCI Plug and Play BIOS in Flash EPROM. Intel and Chips and Technologies provide the Intel/Chips and Technologies 69000 HiQVideo BIOS and controller with 2 MB of integrated SDRAM. Physical video can be a standard CRT, a flat panel, or both. The VGA BIOS is integrated in the system ROM. Up to 128 MB of unbuffered 3.3-V SDRAM can be installed. It is equipped with 32 MB.

Silicon Disk is the new rage in embedded, and the SBC-MediaGX does it's share to support it. The SBC-MediaGX that was delivered contains 8 MB of Intel Strata flash memory. My SBC-MediaGX is half-full (or half-empty), because you can top it out at 16 MB of flash memory with this unit. A CD with Datalight FlashFX file system is also included with the SBC-MediaGX kit.

On the spinning side of disk storage, the SBC-MediaGX maintains all of today's common floppy drives and supports PIO Mode 4 or Ultra DMA/33 Hard Disk and ATAPI CD-ROM. The enhanced IDE subsystem operates in bus-mastering mode with a maximum of two devices (see Photo 1).

Integrated I/O is a healthy part of the SBC-MediaGX. There's an NS97317 with built-in real-time clock and keyboard controller and, for applications that require audio, Cyrix's XpressAAUDIO 16-bit SoundBlaster-compatible subsystem is included. There are even line-in, line-out, and microphone ports.

To fill out an already impressive array of functions, the SBC-MediaGX sports a high-speed parallel port that can be configured via BIOS for SPP,

Photo 1—*Just think, someday all PCs will be dense packages like this.*

EPP, or ECP mode. Four 16C550-compatible high-speed UARTs capable of supporting RS-232/-422/-485 complete the SBC-MediaGX's integrated I/O package.

WHAT'S IN THE DRIVE?

Why it's Windows NT 4.0 Embedded, and not too much at that. The spin of Windows NT 4.0 Embedded that is loaded in the M-Systems flash disk is a minimal configuration of WinNT, which is about 20 MB of code. This means that there is absolutely nothing but the bare necessities of NT on the SBC-MediaGX. This is the way you want to see it. If you can do things with minimal configuration, just think of the possibilities that could exist with a larger component load. Here's the step-by-step account of my initial contact with this particular Arcom blue box.

TO THE FLORIDA ROOM

I've been through the blue box parts ID process with little change from time to time. So, I naturally assumed that the power brick that comes with the SBC-MediaGX would be all I needed to get some pixels to appear on my CRT. Again, I looked carefully at the orientation of the power brick connector and smiled because I can put it on either way without fear of smoking the SBC-MediaGX system.

At this point, I did not know what to expect with the OS or application and was anxious to see what was in store this time around. So with that, I proceeded to fire it up, only to find….

"Disk Boot Failure." How can this be? The only floppy or traditional hard drive attached was that tiny M-Systems product which should have something, anything, on it. This made me wonder what I'd missed. Before I made any more assumptions, I went back to the blue box to see if there was anything in it that could help me.

There I came upon another power connector that links to the SBC-MediaGX at one end and a standard red-black-black-yellow +5 /GND/GND/+12 PC connector at the other. It took a couple minutes in the woods

Photo 2—*10.10.0.1 is also known as ppl_pdc. That's short for PIC Propulsion Labs_Primary Domain Controller. You guessed it, there's a back-up domain controller in the Florida room, too (ppl_bdc).*

after the sun went down before it hit me! There's a PC-type power connector on the M-Systems flash disk. In the South we'd say "It ain't got nothing connected to it," meaning the connector was empty. When I took a closer look at the power brick and noticed it had no connector scheme that would interface to both the SBC-MediaGX and the M-Systems flash disk, I knew I would need a real PC power supply to make this bird sing.

I keep a loose PC supply around for emergencies such as this. I hooked up the PC supply to the SBC-MediaGX and M-Systems flash disk and fixed that pilot-induced error. Everything worked better from then on with power applied to everything that required it.

Now that I have this desktop at my disposal, what can I do with it? Lately, I've been generating waveforms and timings for all sorts of things. Let's jump to some data moving using only the SBC-MediaGX parallel port, Bill's VB6, and some Ryle Design ExacTicks DLL routines.

IS IT REAL?

I had to connect a PC power supply anyway and, because I had a floppy power connector handy, I thought I'd hook up a floppy drive to see if I could get enough timing code to do something useful. Think again! Seems my custom version of WinNT was built without a floppy interface driver. That's fine, because when it comes to the drudgery of moving data and files, I call in WinNT's network services.

Throughout this and other articles, I've addressed how the embedded version of WinNT 4.0 is exactly like the commercial version you can buy on CD. Well, in this case, that's good because I'm going to have to use the WinNT networking services of WinNT Embedded 4.0 to get my SBC-MediaGX timing application code from a WinNT 4.0 server in the Florida room.

In this predicament, the good news is that the SBC-MediaGX is equipped with onboard Ethernet capability and the preloaded kernel I received is set up as a DHCP client. For you future MCSEs, that means the SBC-MediaGX/WinNT embedded combination will ask for an IP address from the DHCP server in the Florida room lab using a standard Ethernet connection.

The DHCP server is set up to service an internal 10.10.0.0 network subset masked as 255.255.0.0 with 11 DHCP available leases. Photo 2 shows the Windows-related technical details of how this is set up. When the SBC-MediaGX fires up WinNT, the DHCP client on the SBC-MediaGX requests a lease on an IP address from the

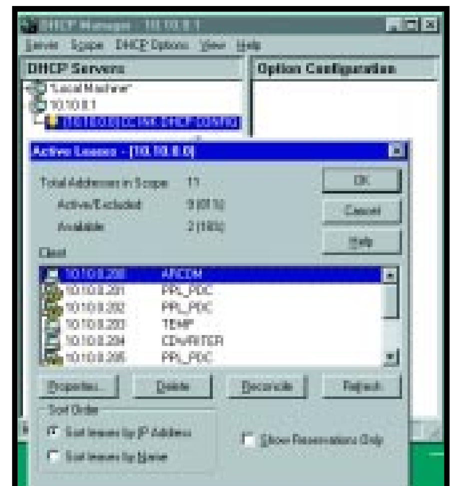


Photo 3—*I took a count of how many PCs versus leases are on this network segment. There are only eight PCs, three using DHCP. The other leases are used by the server's comm ports.*

WinNT server. As you can see in Photo 3, the Arcom SBC-MediaGX successfully requested and leased the IP address 10.10.0.200. This is a good thing because now there's a link for communication between the Arcom SBC-MediaGX and the WinNT server.

Because I don't have any way of getting data into my SBC-MediaGX using a local spinning disk, I'm forced to use the Florida–room network and its resources. Although NetBEUI and NWLINK are installed on the 10.10.0.0 network, they are not installed on the Arcom SBC-MediaGX, but as you can tell by the dot-addressing scheme, good old TCP/IP is everywhere. If TCP/IP is around, there's a chance that FTP or TFTP may be hanging around with it, but before you can get the data load from the WinNT server, you've got to grease the skids a bit.

Take a look at Photo 4. Here I used Microsoft Internet Service Manager to designate which drives the Arcom SBC-MediaGX and WinNT Embedded 4.0 can use FTP against. At this point, I had to transfer the ExacTicks software from a floppy on the WinNT server. So, not knowing what I needed or where it would come from, I included the WinNT server's hard disk (c:) and CD-ROM (d:) drives as FTP targets, too (see Photo 5). Users are already set up on the network who should be able to use FTP via the Arcom SBC-MediaGX.

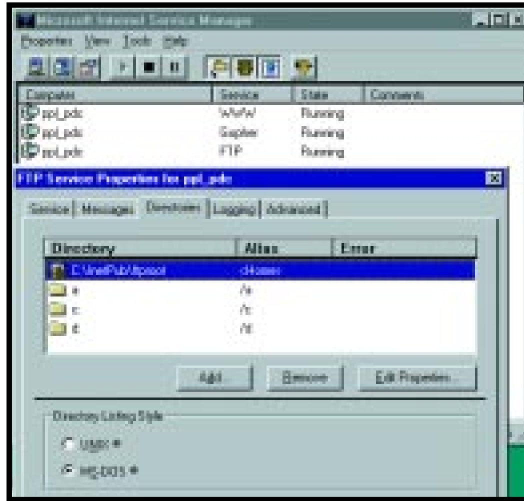


Photo 4—*We only need to get to the floppy drive, but what are a few extra keystrokes among friends?*

A quick FTP-initiated ls command directed at the WinNT server floppy drive produces an entry for the target ExacTicks program named `setup.exe`. A WinNT server side confirmation of the Arcom SBC-MediaGX FTP session is shown in Photo 6. All that's left is to get the bits from the WinNT server floppy disk to the SBC-MediaGX's flash disk.

Referring to Photo 7, after a change to binary transfer mode, it seems that the FTP session was successful and the ExacTicks data did get loaded into the Arcom SBC-MediaGX. Victory!

IT'S ABOUT TIME

ExacTicks is a collection of high-precision timing libraries that can be accessed using the Bill, Watcom, or Borland C compilers, or Delphi or VB. ExacTicks is an indispensable tool for generating precision pulse trains and delays.

ExacTicks relies on the timing mechanism built into Windows. ExacTicks can be integrated into both 16- and 32-bit Windows applications. This includes Win3.x/95/98/NT. Sixteen-bit applications are defined as Win3.1 or WFW (Windows For Workgroups). Native and 32-bit applications are those running natively with Win9X and WinNT.

ExacTicks uses a timestamp as its fundamental time measurement component. This retrieves a 64-bit count from the host system's timer controller. An ExacTicks timestamp has nothing to do with the host system's time-of-day-clock.

Timers, delays, alarms, events, and reports are ExacTicks components. ExacTicks timers are like stopwatches and return elapsed times. Under WinNT, a timer may also return a value that is relative to the CPU time consumed by the host process. ExacTicks delay components pause execution of the current application for a specified interval. The executing code's control is passed to the delay for a specified interval.

Listing 1—*This is a great tool for people who need guidance using C. I wanted to show the* `include`*s here.*

```
** Find the correct parallel port **
findport3bc:
caddr = &H3BE
Out caddr, &H8
sbyte = Inp(caddr - 1) And &H80
If sbyte = &H80 Then GoTo test3bc
GoTo findport378
test3bc:
Out caddr, &H0
sbyte = Inp(caddr - 1) And &H80
If sbyte = 0 Then GoTo foundport
findport378:
caddr = &H37A
Out caddr, &H8
sbyte = Inp(caddr - 1) And &H80
If sbyte = &H80 Then GoTo test378
GoTo findport278
test378:
Out caddr, &H0
sbyte = Inp(caddr - 1) And &H80
If sbyte = 0 Then GoTo foundport
findport278:
caddr = &H27A
Out caddr, &H8
sbyte = Inp(caddr - 1) And &H80
If sbyte = &H80 Then GoTo test278
GoTo noportfound
test278:
Out caddr, &H0
sbyte = Inp(caddr - 1) And &H80
If sbyte = 0 Then GoTo foundport
GoTo noportfound
foundport:
daddr = caddr - 2
saddr = caddr - 1
init:
Select Case daddr
Case &H3BC
   lblport.Caption = "Connected to Parallel Port at Address 0x3BC"
   'lblstatus.Caption = "Port 0x3BC"
Case &H378
   lblport.Caption = "Connected to Parallel Port at Address 0x378"
   'lblstatus.Caption = "Port 0x378"
Case &H278
   lblport.Caption = "Connected to Parallel Port at Address 0x278"
   'lblstatus.Caption = "Port 0x278"
End Select
** CALIBRATE TIMER ROUTINES AND SETUP MINIMUM DELAY**
delaycal = hrt_delay_alloc(500, HRT_MILLISECOND)
mindelay = hrt_delay_getmin
Select Case mindelay
   Case Is < 20
      usedelay = 20
   Case Is > 20
      usedelay = mindelay
End Select
hrt_delay_free (delaycal)
delayuse = hrt_delay_alloc(usedelay, HRT_MICROSECOND)
** BIT BANG SUBROUTINES **
clklowdatalow = &HC3
clkhighdatalow = &HCB
clklowdatahigh = &HD3
clkhighdatahigh = &HDB
Public Sub highout()
         Out daddr, clklowdatahigh
         hrt_delay_do delayuse
         Out daddr, clkhighdatahigh
         hrt_delay_do delayuse
         Out daddr, clklowdatahigh
End Sub
Public Sub lowout()
         Out daddr, clklowdatalow
         hrt_delay_do delayuse
         Out daddr, clkhighdatalow
         hrt_delay_do delayuse
         Out daddr, clklowdatalow
End Sub
```

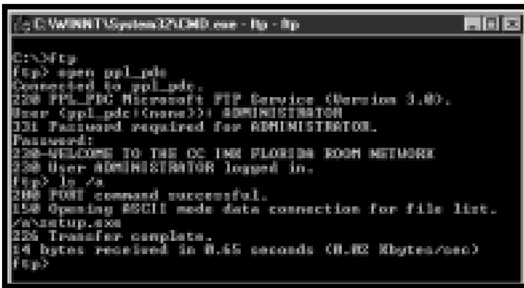


Photo 5—*There's nothing special here. Note that anonymous spectators are stopped at the gate.*

Alarms are software components that are set for a certain time interval and are not asynchronous. Thus alarms must be checked periodically. An ExacTicks alarm will not trigger other actions. It will simply notify the process, checking that the interval has come and gone. The other side of an ExacTicks alarm is an ExacTicks event. An ExacTicks event is similar to a DOS interrupt service routine, like a Windows Callback function. An ExacTicks report is just that, a report—a summary of active and named timers, activations, elapsed time, and average activation time.

To summarize, if you only want to delay and not perform any other functions, use the delay routines. If you need to go about your business while a delay is processing, and you have the overhead to check up on the elapsed time, use an ExacTicks alarm. If you don't have to check overhead time and need asynchronous signaling, use an event. A report is still a report no matter what you need to do.

If the application was running under DOS, the precision time measurement would be completed by directly addressing the 8253/8254 timer counter IC (or its look-alike) on the embedded PC. As you know, you can't do that with Bill's Windows products. When running Windows, you still get the count from the timer IC, but you must retrieve it in a roundabout way.

In a 16-bit Windows environment, INT 0x2F gets the Virtual Timer Device (VTD) function dispatch address. Calling this address with AX = 0×100 returns a 64-bit count from the VTD. For the 32-bit world of Win9X and WinNT, there are two API functions, `Query-PerformanceFrequency()`, which returns the frequency of the system's precision timer and `QueryPerformanceCounter()`, which registers a 64-bit count from the timer.

That's sufficient if all you need is a raw 64-bit count, but to make the count information useful, you must also consider the time it takes to get these counts and factor that into your precision timing routines. To get precise timings without rewriting and reinventing timing code for every project, you must produce routines that are self-calibrating at run time and share a common API for both 16- and 32-bit environments. Without elaborating about how this is done, I'll just say the ExacTicks DLL is the ticket here.

TIMING IS EVERYTHING

I've been exploring ways to get data to and from embedded systems. I've used the parallel instead of the serial port to interface to projects like parallel-port-based PIC programmers and flash-laden ICs on Internet appliance prototypes. This reduces hardware and compensates for code by adding software overhead.

The eliminated serial hardware costs about $10 in single quantities. The extra software costs next to nothing in any quantity.

It's ironic to use the Arcom SBC-MediaGX's parallel port because it's loaded with serial interfaces, but you may find this interesting. It may be handy if your project is strapped for serial resources, or if you want to reduce project parts and save bucks or board space. To this end, I'll describe a simple project that employs the ExacTicks DLL routines to produce a precision set of bit-bang pulses from the SBC-MediaGX's parallel port.

PRECISION BIT BANGING

There are 47 pages of ExacTicks command syntax. Obviously, I won't expound on each and every detail, but I will use Listing 1 to describe and demonstrate the ExacTicks functions that are necessary to carry out the parallel–port mission. Believe it or not, you can do this whole trick using only delay routines. The SBC-MediaGX is your intended target, but I'll show how ExacTicks routines make this application universal.

Not all embedded platforms are lighting-fast, so you must compensate for PCs that run at low clock speeds. I set a low point of 500 ms as the widest programmed pulse that can leave a pin from the parallel port. Conversely, 20 µs is the shortest pulse duration I'll generate. The selection of these values depends on how quick the equipment is at the other side of the parallel port and your embedded PC's speed.

In this case, the equipment on the other side of the parallel port is a PIC running at 10 MHz. I chose these values so my entire PIC bit-bang algorithm could easily recover bits and do housekeeping within the minimum pulse width of 20 µs. As for the embedded PC, there should be no problem generating a 500-ms pulse at the lowest of practical clock speeds.

Before any bit banging or timing takes place, it would be nice to know which parallel port you will be operating on. My port-finding code is shown at the beginning of Listing 1, and the algorithm depends on a short between an output pin and the busy input pin on the external equipment parallel port interface. If the external equipment is hooked up, a TTL high and low are sent out and sensed accord-

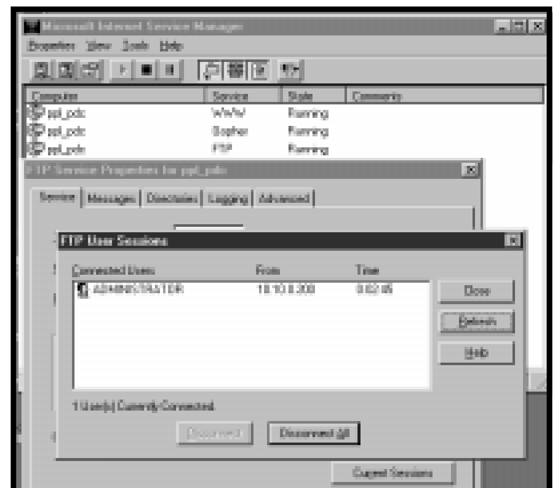


Photo 6—*Having your own private network has its privileges. Administrator is king.*

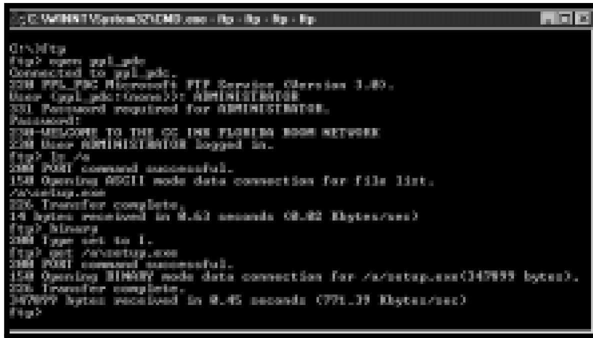


Photo 7—*Verbose Mode doesn't lie. As you can see, I tried unsuccessfully to do an ASCII transfer on a binary file.*

ingly using the busy input via the short on the external equipment's parallel interface. After the parallel port control port address is determined, the data and status port addresses can be easily calculated.

When the parallel port is established and the external equipment is sensed, the timing variables can then be initiated. The first step is to allocate a delay timer. This is done using the `delaycal = hrt_delay_alloc(500, HRT_MILLISECOND)` statement. `Delaycal` has significance here because the first time a timer is allocated, ExacTicks calibrates its timing routines automatically. So, by simply allocating a 500-ms timer, you calibrated your software routines and allocated the maximum delay timer in the same motion.

Next, determine if the embedded PC can handle generating a 20-µs pulse. The `hrt_delay_getmin` function shows that the minimum delay time in microseconds is returned as `mindelay`. If the minimum delay is less than or equal to 20 µs, use 20 µs as the value. If the minimum delay returned is greater than 20 µs, use the returned value. Although you allocated a 500-ms delay called `delaycal`, the application doesn't need it, so free its resources using `hrt_delay_free(delaycal)`. Then, use the delay allocate routine to create a delay called `delayuse` that reflects the 20-µs default or larger `mindelay` value.

Now that the delay is calibrated and allocated and the parallel port address is known, put the data on the parallel port pins. The final lines of Listing 1 show how this is accomplished using the delay use timer you allocated earlier. The clk *xxxxxxx* statements are for clarity and are constants defined elsewhere in the code. You know where `daddr` came from and the out command is a user-written I/O DLL, as VB doesn't have a native in or out instruction. Here's the decode on the clk *xxxxx* data statements:

```
clklowdatalow   11000011
clklowdatahigh  11010011
clkhighdatalow  11001011
clkhighdatahigh 11011011
```

Without the eternal 1s and 0s, bit 5 is the data bit and bit 4 is the clock bit. Because a clock bit is present, this is a clocked synchronous bit bang.

TIMED OUT

I've cashed out all my Listings, Figures, and Photos, but note there are 20-µs clock and data pulses on the SBC-MediaGX parallel port.

That wraps up my coverage of Windows NT Embedded 4.0 for a while. If Microsoft decides that I need a full copy, I'll be back to share it with you and again prove that it doesn't have to be complicated to be embedded. ■

Fred Eady has more than 20 years of 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

SBC-MediaGX
Arcom
(888) 941-2224
(816) 941-7025
Fax: (816) 941-7807
www.arcomcontrols.com

Flash FX file system
Datalight
(360) 435-8086
Fax: (360) 435-0253
www.datalight.com

FEATURE ARTICLE

Shawn Arnold

Implementing a RAS Server Port

In the February and March issues of Circuit Cellar Online, Shawn covered the background information on designing a DSP-based RAS server. This month he enlightens us on the basic but critical issues regarding RAS port choice.

As a result of today's demand for Internet and remote LAN access, the RAS (remote access server) market is growing quickly. For success in the competitive market, manufacturers must design and build high–performance, inexpensive products. Hardware and software design engineers must design flexible, expandable, and enduring products.

One important design issue is integrating the RAS product's heart—the RAS port. Today's products offer a range of choices. But, you must understand subtle issues regarding ports or risk poor system performance.

This article will present important issues regarding RAS port choice and usage to help you get optimal results. The ADSP-mod870-100 Internet Gateway Processor is a good place to start. I will discuss how to integrate the mod870 RAS port into a RAS server. I'll explore hardware integration, which involves connecting signals among the mod870 RAS port and the Telco and Network interfaces, and software integration, which involves the host's code routines that allow it to communicate with the RAS port.

HARDWARE DESIGN

One of the first steps of hardware integration is placing the RAS port into the RAS server's data stream, which flows between the remote user and the network (see Figure 1). The RAS port becomes the gateway by which a remote user can connect and gain access to a given network. The RAS port sits between the server's Telco and Network interfaces.

The hardware design consists of the physical connections among the RAS port and the RAS server's Telco interface on one side, and the Network interface on the other. Note that the Telco interface is a serial connection, and the Network interface is a parallel connection.

Remember that you are working within a RAS server, hence, integration involves the connection of multiple port devices to both the Telco and Network interfaces. Collectively, this group of RAS ports is called the RAS port bank.

The Telco interface (see Figure 2) involves the connection of the RAS server to the telephone system. The most efficient way to feed the RAS server is through a digital trunk line. Examples of digital trunk lines are the U.S. standard, T1, European standard, E1, and ISDN PRI.

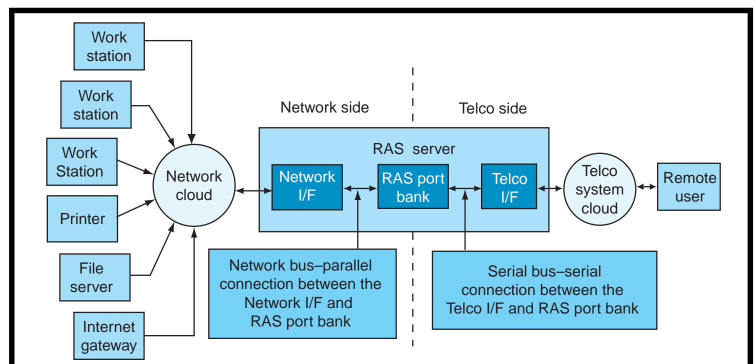


Figure 1—*One of the first steps of hardware integration is placing the RAS port into the RAS server's data stream, which flows between the remote user and the network.*

A few years ago, a RAS server may have consisted of a box with multiple RS-232 lines connected to racks of analog modems, with each modem fed by its own analog local loop. But today, RAS ports enable efficient integration of the modem or port device into the RAS server box. Now, you can use a single digital trunk line that carries multiple calls on one pair of wires.

Port devices must connect to the serial TDM signal PCM datastream, which is carried by the digital trunk lines. Hence, each port device in the RAS server must have access to the PCM datastream. For this reason, the serial connection is bused to each port device in the bank.

A Telco serial connection usually is straightforward because a special port device selection isn't required. A port device is assigned to a channel in the serial TDM signal. During its assigned time slot, the port device senses in the receive direction and drives in the transmit direction.

The Network interface involves the connection of the RAS server to the network (see Figure 3). The host handles the network data that flows between the RAS port and the network, processing network packets as needed. The host supports a bank of port devices. Depending on its horsepower, the host supports a number of ports (the amount is a multiple of the number of channels carried on the digital trunk line). Usually, it connects to a parallel port on each port device in the port bank.

Decoding the signals that select which port device the host communicates with is a challenge. Unlike the Telco side, where each port device is assigned a time when it can use the serial bus, the host controls the use of the network bus that is shared by all of the ports.

It controls the parallel port, bus, and the operation of each port device individually. This usually includes driving reset signals to and sensing interrupt signals from a port device. Other possibilities include driving power-down signals, driving interrupt signals, and driving timing signals to the port device.

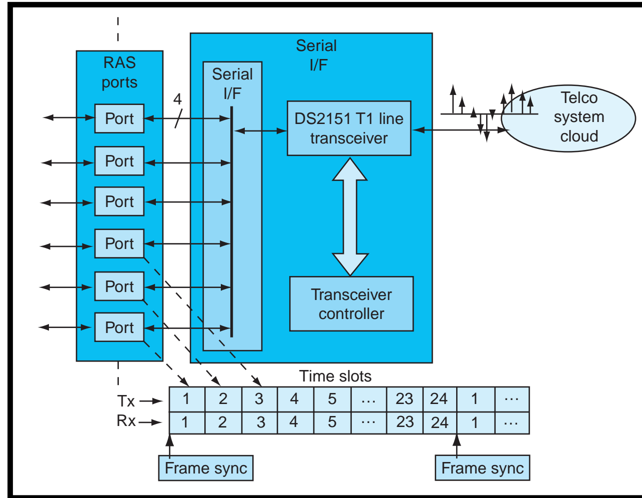


Figure 2—*The Telco interface involves the connection of the RAS server to the telephone system. Feeding the RAS server through a digital trunk line is efficient.*

SOFTWARE DESIGN

Because the RAS port is usually supplied with modem, fax, and voice firmware, the software design does not require code written for the port device. The port is viewed as a black box, so the focus is on code development for the host's support of the RAS port. The host software design is comprised of hardware abstraction and an API interface (see Figure 4).

Hardware abstraction, or hardware driver, develops the lowest level function calls that allow data buffers to transfer between the host and port device. Code routines are created for the functions that the physical host-to-port device connections can perform. The functions are combined to form an abstraction that removes the host code development from the specific aspects of the physical hardware connection.

Like a carpenter, you follow a blueprint, which enables the host to communicate with the firmware that runs on the RAS port. The blueprint, called the API specification, describes how the communication mechanism works and how the host can communicate with the RAS port firmware. The API interface involves the development of the communicative host routines. Hardware abstraction involves operating port device hardware, and API interface operates RAS port firmware.

THE MOD870

The mod870 RAS port meets the port device specifications. Remember that the two important RAS port specifications are small size and low power consumption (in milliwatts). The most important features are universal port and complete solution. And, each RAS port product will have extra features.

The integrated SRAM and I/O peripherals (SPorts and IDMA port), and low pin count contribute to the mod870's small size. The 0.4µ process geometry, 40-MHz clock rate, and idle and power-down states reduce its power consumption. And, it is available with firmware that implements the three applications that make up a true universal port:

- modem: V.90, K56 flex, V.34, V.32/bis, V.23, V.22/bis, and V.21
- fax: T.30 class 2.0
- voice: G.711, G.722, G.728, G.723/.1, G.729/a, and G.165 EC

The mod870 offers a complete solution because port device programming isn't required, and applications

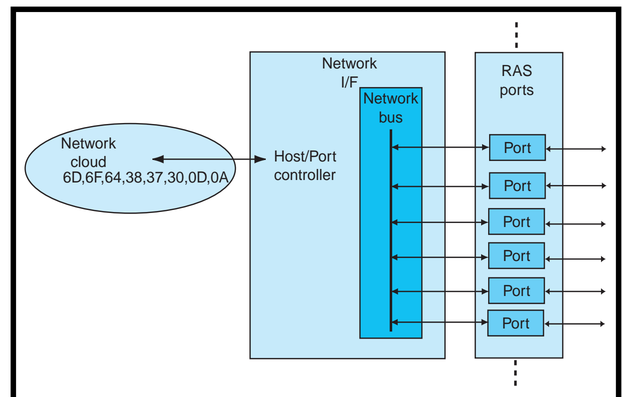


Figure 3—*The Network interface involves the connection of the RAS server to the network.*

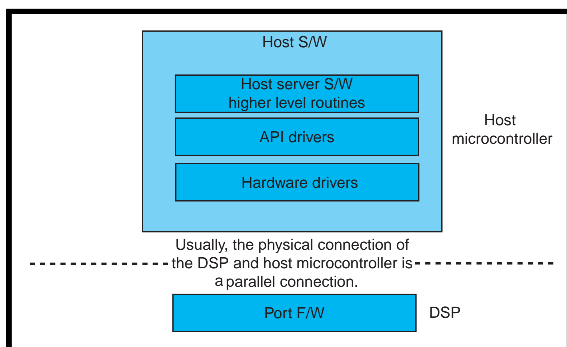


Figure 4—*The host software design is made up of hardware abstraction and API interface.*

don't need additional coding. Imagine the port as a black box. The API specification describes how to configure, control, and communicate with each of the universal port applications that run on the mod870 RAS port.

THE PHYSICAL CONNECTIONS

Now, let's discuss how to integrate the mod870 RAS port into a RAS server. Each RAS port must be connected to the digital trunk line to support its channels. Often, the connection is serial because the line carries a serial TDM signal. Because a digital trunk line carries many simultaneous calls, it usually is bused to a bank of RAS ports. The number of ports in the bank equals the number of calls carried by the digital trunk line. Here are three common digital trunk lines: T1 carries 24 calls, E1 carries 30, and ISDN PRI carries 23.

In my example, the RAS server is specified to support a T1 digital trunk line. A Dallas DS2151 T1 transceiver implements the transceiver circuit. This transceiver provides the serial bus connection to a bank of 24 mod870 ports (see Figure 2).

The mod870 has two integrated serial port peripherals, known as SPort0 and SPort1. One is needed for the RAS port application. SPort0 is used by the RAS port firmware because it has a 24/32-channel TDM mode of operation. This allows easy connections to T1 or E1 digital trunk lines. In TDM mode, SPort0 has a four-pin interface (see Table 1).

The mod870's firmware offers configurable modules that perform generic PCM data stream processing functions. They allow the mod870 to be connected to transceiver circuits other than the DS2151. These modules include data companding, U-law or A-law, and bit reversal. The digital trunk line, Telco interface transceiver, and supported data format determine the combination of PCM datastream processing functions configured and enabled. This provides the proper data format to the port's internal modem data–processing modules.

The mod870's SPort0 connection to the serial port of the Dallas 2151 T1 transceiver is shown in Figure 5. Notice the simple four-wire serial connection of the Telco interface and mod870's SPort pins. These four lines carry the PCM datastream between the digital trunk line transceiver and the port bank. Because the serial lines carry 24 TDM channels, the serial lines form a bus that connects to all 24 mod870s in the port bank.

In each mod870, SPort0 is configured for multi-channel mode, so only one frame sync line is needed. The frame sync connects to the SPort0's RFS0 signal. In the multi-channel mode case, the TFS0 signal becomes the TDV output signal and isn't used.

The serial clock signal, or SCLK0, is driven by the DS2151 and sensed by SPort0. It provides the master clocking that drives the SPort0 operation.

The frame sync signal, or RFS0, is driven by the DS2151 and sensed by SPort0, too. It provides the start-of-frame reference point. The PCM datastream is a 24-channel frame serial TDM signal. The RFSO provides the reference point that marks the start of each 24-channel frame.

Pin name	I/O	Function	Comments
SCLK0	I/O	serial clock	bussed, configured as an input
RFS0	I/O	frame sync	bussed, configured as an input
DT0	O	transmit data	bussed, active on one channel
DR0	I	receive data	bussed, active on one channel

Table 1—*In TDM mode, the mod870's SPort0 has a four-pin interface.*

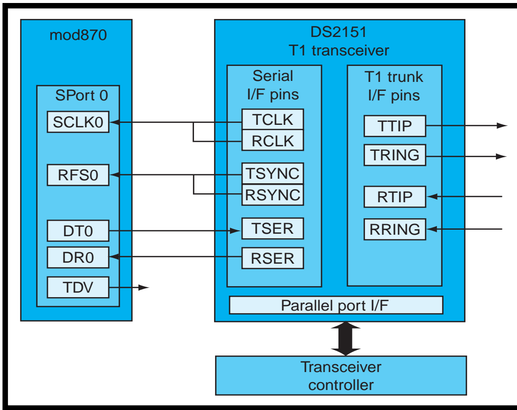


Figure 5—*Here's the SPort0 connection to the serial port of the Dallas 2151 T1 transceiver.*

The transmit data signal, or DT0, is driven by SPort0 and sensed by the DS2151. It carries the mod870's transmitted PCM datastream. The downstream PCM flows from the network to the host, then to the mod870. The firmware application modulates the network data and transmits it via SPort0 to the remote device. The DT0 is bused to each mod870 in the port bank. One mod870 is assigned a given channel at a time and drives it in the serial TDM signal.

The receive data signal, or DR0, is driven by the DS2151 and sensed by SPort0. It carries the mod870's received PCM data stream. The upstream PCM data flow is received from the remote device via SPort0 into the mod870. The mod870 firmware application demodulates the Telco data and passes it to the host, then the host passes it to the network. The receive data signal is bussed to each mod870 in the port bank. Only one mod870 is assigned a given channel at any time, and senses it in the serial TDM signal. The PCM datastream on a T1 digital trunk line is U-law companded. Therefore, the U-law companding module within the mod870's firmware application is enabled.

THE NETWORK INTERFACE

To support the network data flow, each port device must be connected to a controlling host. The host usually supports a bank of port devices. In most cases, the host supports a multiple of the total number of calls carried by the digital trunk line.

Most port devices have a parallel port for connection to a controlling host device. For this reason, the connection between the host and the port is made between the host's memory interface and the port device's parallel port. Unlike the serial interface bus, the Network interface bus is not TDM'd. Transfers over the network bus are asynchronous and occur randomly, therefore the host controls and arbitrates this bus. To select individual parallel ports for data transfer, each select signal is mapped into the host's external memory space.

In this example, the host supports 24 mod870 ports, which matches the number of channels carried by the T1 digital trunk line (see Figure 3).

The mod870 includes an integrated internal direct memory access (IDMA) port. The IDMA port is designed for connection to a master host device. It is a 16-bit parallel port that supports

mod870 booting via the host and runtime access to mod870 internal memory. And, it has a 16-bit multiplexed address and data bus. The multiplex bus adds complexity to the host interface, however, the pin count savings contributes to a reduction in package size.

IDMA port access is asynchronous and a host can access the mod870's internal memory while the mod870 is operating at full speed. It does not require core intervention to maintain data flow. The host system can access mod870's internal memory directly. The mod870's IDMA port has a 21-pin interface (see Table 2).

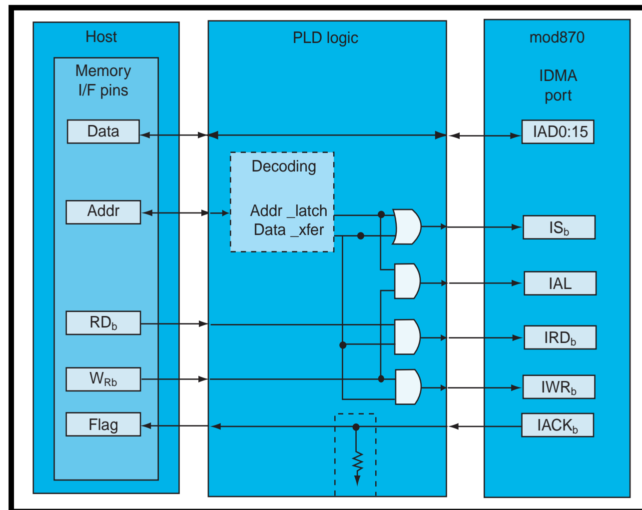


Figure 6—*The mod870's IDMA port connections to the host's external memory interface are complicated by the IDMA port address and data signals, which are multiplexed onto a single bus.*

CONNECTING THE IDMA PORT

The mod870's IDMA port connections to the host's external memory interface are complicated by the IDMA port address and data signals, which are multiplexed onto a single bus (see Figure 6). More address decoding, in addition to port device selection, must be implemented to distinguish an address latch from a data transfer cycle.

Because the host individually selects the 24 ports, all but the IDMA port IS_b signals are bused. So, a single host address is allocated for each address latch and data transfer select on each of the IDMA ports.

The host accesses the appropriate memory-mapped address to select the operation that it wants to perform. The required address locations for 24 mod870 ports are:

N modems → 2 × N address locations
24 modems → 2 × 24 = 48 address locations

The required number of decoding bits per address location for 24 mod870 ports are:

N modems → int (log2 [2 × N] +1) address bits
24 modems → int (log2 [2 × 24] +1) = 5 address bits

[*IS] selects the IDMA port for access. A mod870's IDMA port will be accessed for either an address latch cycle or a data transfer cycle. Because the host supports multiple mod870s, each pair of accesses to a given mod870 IDMA port must be unique. Each pair is given a unique address in the host's external memory space.

Every *IS pin on each mod870 has a separate connection to the decode logic that decodes the host's memory-mapped address. A mod870 IDMA port should be selected whenever an address or data cycle is desired for that mod870 IDMA port.

[IAL] latches a mod870 address into the IDMA port. This signal is active during an address latch cycle. Note that a mod870 IDMA port must be selected via its *IS pin or signal in order for an address to be latched to the IDMA port. So that, the port bank's IAL pins can be bused. Any host write to an address latch mapped location of any mod870 IDMA port will activate the signal. However, only one *IS signal should be asserted.

IDMA Write Strobe, or IWR_b, enables an IDMA port data write. This signal is active during a data write cycle. Note that the IDMA port must be selected via its *IS pin or signal in for a write to mod870 memory to occur. For this reason, the port bank *IWR pins or signals can be bussed, provided each *IS signal is unique to each mod870 IDMA port. Any host write to a data transfer mapped location will activate this signal.

IDMA Read Strobe, or *IRD, enables an IDMA port data read. This signal is active during a data read cycle. The IDMA port must be selected via its *IS pin or signal for a read from mod870 memory to occur. So, all the port bank *IRD pins or signals can be bussed if each *IS signal is unique to each mod870 IDMA port of the bank. Any host read to a data transfer mapped location will activate this signal.

The IDMA Port Acknowledge, or *IACK, line identifies the completion of a data transfer cycle. It acts as a busy signal for the IDMA port transaction. The host waits for this to be inactive before starting the next IDMA operation.

The *IACK pin can be configured as an open drain output. This allows all the *IACK signals from the bank's ports to be wire ORed or bused, even though this is an output signal. This helps reduce the complexity of the connection back to the host. Only one host connection is required. It could be a MACK (memory acknowledge) or a host input flag pin, which may be required to extend the access cycle.

The multiplexed address and data bus (IAD[15:0]) carry the address for a data transfer during the address latch cycle and the data during a data transfer cycle. The host's data bus is connected to this bus. The host's address bus drives the decode logic, which selects the desired device by asserting its *IS signal when that device's address location is accessed.

THE CONTROL INTERFACE

The host also drives and senses general control signals to and from port devices. The signals include:

- reset—port device input, driven by the host
- flags—host input, driven by the port device
- interrupts—port device input, driven by the host

Pin name	I/O	Function	Comments
IRD_b	I	Read strobe	bused
IWR_b	I	Write strobe	bused
IS_b	I	Port select	separate connection per port
IAL	I	Address latch	bused
IAD [15:0]	I/O	Address/data	bused
$IACK_b$	O	Acknowledge	bused, open drain output

Table 2—*The mod870's IDMA port has a 21-pin interface.*

The host should have individual control of each port device's reset signal, because port devices crash sometimes despite safety measures. Having individual control of the reset lines allows a specific port device to be reset, rebooted, and restarted independent of other port devices in the port bank. If the ports shared a common reset, the whole port bank would have to be reset.

Some port devices provide signaling flags that mark the occurrence of events within them. The occurrence of one of these events may require the host to perform a specific action. The most common use of this signal is to generate a hardware interrupt signal to the host. For example, a port device may pulse a flag pin, indicating the need for the host to read received network data from the port device.

Some port devices require interrupt input signals to be manipulated to activate certain actions within them. With interrupts, the host drives the signals in order to elicit the proper operation from the port device. For example, the host has to signal the port device that it completed the write of network data into the port device for transmission.

THE MOD870'S CONTROL SIGNALS

The mod870 has two useful control signals (see Table 3).

Although not required, additional connections can be made between the host and other control signals. This increases individual control over each mod870 and adds flexibility and efficiency to the system. The mod870's control signal connections to the host are shown in Figure 7.

In my example, the host controls 24 mod870 ports. In order to provide the greatest flexibility in mod870 control, the design requires individual control over each mod870 reset signal. This individual control requires decoding logic. Decoding is similar to IDMA port select signal decoding. Each reset is allocated a unique selection address in the host's I/O memory. Every *RESET pin on each mod870' has a separate connection to the decode logic, which decodes the host's memory/mapped address and activates the mod870's reset. In this case, each signal is assigned an address. The host addresses the appropriate select depending on the desired signal operation. The required address locations for 24 mod870 ports are:

N modems → N address locations
24 modems → 24 address locations

The required number of decoding bits per address location for 24 mod870 ports are:

N modems → int (log2 [N] +1) address bits
24 modems → int (log2 [24] +1) = 5 address bits

Because there are fewer interrupt input pins on the host than FL0 signals coming from the port bank, the FL0 signals must be decoded to one signal going to a single host interrupt line. In this case, there are 24 FL0 signals coming from the port bank. Notice that when the host senses an interrupt, you don't know which mod870 generates the signal. It is determined by using a 24 input OR gate and a single 24-bit memory mapped register. The OR gate is used to decode the FL0 signals to the single host interrupt signal. The 24-bit memory-mapped register captures the individual toggles of the 24 mod870 FL0 pins. When the host is interrupted, it can read the memory-mapped register for the information.

You may want to mask out certain mod870 FL0 signals. For example, if a polling scheme is employed for mod870s operating in fax mode, mask out interrupts generated by the mod870. This feature can be implemented via a 24-bit memory-mapped mask register. The FL0 register bits are AND'd with the mask register bits to generate the inputs to the OR gate. Only the unmasked ports can generate a host interrupt signal. The host can write the mask register with the desired masking value.

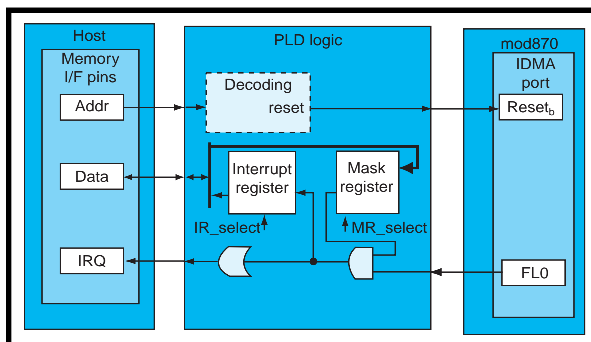


Figure 7—*Here are the mod870's control signal connections to the host.*

SOFTWARE DESIGN: CODING THE HARDWARE DRIVERS

Hardware abstraction removes the specifics of the hardware implementation from software development. The most important reason is code portability. For example, if the system hardware changes, a more powerful port device is integrated in the system, you'll want to minimize the effects on the software. Higher level software coding should never depend on the specifics of the hardware implementation.

If the abstraction is properly executed, changes in the system's hardware result in minor rewriting of the lowest level hardware interface/driver functions. The first step is to clearly mark the boundary of abstraction. In the RAS server host, this boundary is defined by the following function calls:

```
get_port_buffer()
put_port_buffer()
```

All higher level functions are built upon these two basic functions. If the hardware changes, only the code within these two functions will require recoding.

HOW DOES THE MOD870 WORK?

To design and code the best buffer transfer functions, it is important to know how the mod870's memory is structured and how the IDMA port is used to access this memory. The mod870's memory architecture has two separately selectable and addressable memory spaces (see Figure 8). The memory spaces are referred to as program memory (PM) and data memory (DM) spaces. Program memory space is 16k words by 24 bits, and contains either 24-bit instruction code or 16-bit data. Data memory space is 16k words by 16 bits, and contains 16-bit data words.

Although the program and data memory spaces are only addressable up to 16k words, memory overlays or pages provide an additional 16 words of available memory for both PM and DM spaces. The bottom half of PM and the top half of DM have two 8k word overlay pages.

Therefore, only 14 bits are required to address locations of a memory space. However, a 16-bit value is latched into the IDMA port during the address latch cycle (see Figure 9). The two extra bits provide overlay page and memory information that is required to identify unique locations within the mod870's total available memory.

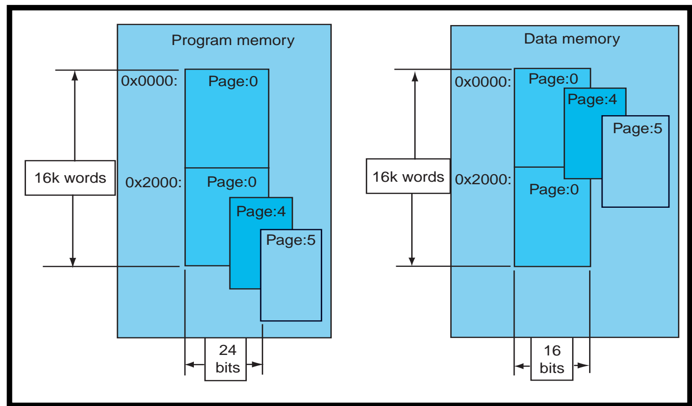


Figure 8—*The mod870's memory architecture has two separately selectable and addressable memory spaces.*

The mod870 IDMA port requires two basic operations, the address latch cycle, which includes overlay page and address latching, and the memory access cycle, which includes read and write accesses. Depending on the value of IAD[15], the address latch cycle will perform one of two possible functions:

- an overlay page latch: IAD[15] = 1
  DM overlay page = IAD[7:4]
  PM overlay page = IAD[3:0]
- an address latch: IAD[15] = 0
  to DM space: IAD[14] = 1
  to PM space: IAD[14] = 0

IRD$_b$ and IWR$_b$ determine the data transfer direction. Note that the IDMA bus is 16–bits wide, therefore PM transfers require two read or write cycles. DM transfers require one cycle. After an address is latched into the IDMA port via an address latch cycle, subsequent data transfers automatically increment the original address value. So, only one address latch cycle is required for a contiguous buffer of data. After the address latch cycles, the buffer can be accessed by continuous memory access cycles. A typical IDMA port access consists of three steps, overlay page latch, address latch, and looped memory access (read or write).

IDMA PORT FUNCTIONS

At the lowest level of host-to-IDMA port interaction, there are five basic transaction cycles the host must be programmed to support. The buffer transfer functions are constructed from these five cycles:

- address latch cycle
- data transfer cycle: 24-bit read—2 cycles/PM word, strobe IRD$_b$
- data transfer cycle: 16-bit read—1 cycle/DM word, strobe IRD$_b$
- data transfer cycle: 24-bit write—2 cycles/PM word, strobe IWR$_b$
- data transfer cycle: 16-bit write—1 cycle/PM word, strobe IWR$_b$

An address latch cycle performs two pointer latch operations that are not distinguished by the way the host controls the IDMA port, but by the address information driven on the IAD bus. Therefore, only one address latch function is required. The address information passed to this function determines the latch cycle, page, or address.

Data transfer cycles can be read or write. And, they can be to or from PM space, requiring two access cycles, or DM space, requiring a single access cycle. These possibilities lead to four required access functions. Then, all data transfers are constructed from the five basic cycles.

It is assumed that `inpw( )` and `outpw( )` are programmed in the assembly language of the specific host platform. The address latches are distinguished by the address value passed to the latch cycle function.

BUFFER TRANSFER FUNCTIONS

Buffer transfer functions are built from the five basic IDMA building block functions.

The IDMA port automatically increments its internal address pointer for each word transferred.

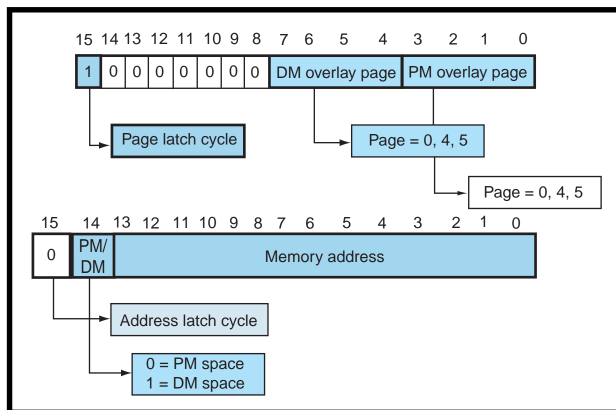


Figure 9—*A 16-bit value is latched into the IDMA port during the address latch cycle.*

Therefore, for contiguous data locations, one address latch cycle is required. For efficiency, the word transfer function is chosen before the buffer transfer loop is executed. This avoids incorporating a redundant "if" statement within the loop.

A transfer register is employed to hold the entire necessary buffer pointer and size information for the transfer. Each item in the transfer register structure must be programmed before buffer transfers. Macros are used to automatically insert the IAD[15] and IAD[14] bit information. This helps the abstraction of the hardware in subsequent higher level functions that will call these buffer transfer functions.

THE API INTERFACE

An API describes the interface to the port device; and the mechanisms by which the host can operate and control the functionality of the RAS port. Generally, API interfaces are reserved for programmable port devices. In the programmable port device case, the API spec describes how the host can interact with the firmware that runs on the port device. The types of interactions may include the following: network data access and transfers, port device configuration and control, data formats, start-up procedures, and initialization.

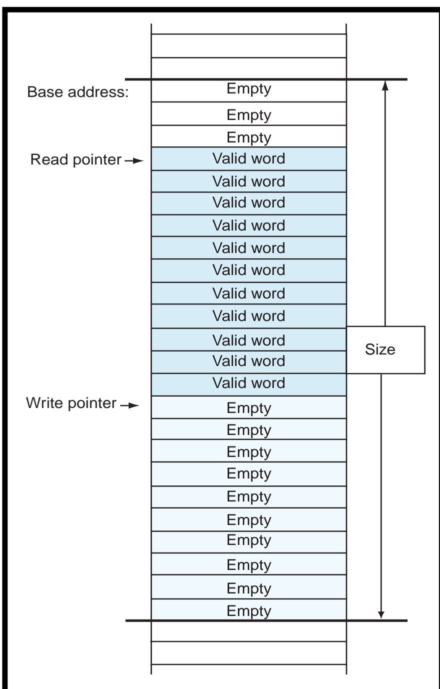


Figure 10—*All data is passed between the host and the modem via virtual FIFOs that reside in the mod870's memory.*

Pin name	I/O	Function	Comments
$RESET_b$	I	chip reset	
FL0	O	output flag	host interrupt

Table 3—*The mod870 has these two useful control signals.*

The API interface refers to the functions that support the data transfer mechanisms described in the mod870's API specification. All data is passed between the host and the modem via virtual FIFOs that reside in the mod870's memory (see Figure 10). The term "virtual" is used because these are not real FIFOs chips. These FIFOs are circular buffers that must be maintained by the host code and the mod870 firmware.

A virtual FIFO is a data buffer with associated read and write pointers. The pointers point to the start and end of the valid data range within the FIFO. The way the pointers are used makes the linear buffer look like a circular buffer. Pointers must be maintained manually. The host code and the firmware update pointers as they read or write the FIFOs and monitor pointers for wrap around when they reach the end of the buffer.

Two functions are created to perform reads and writes to a FIFO. The use of transfer registers helps hide the details of handling PM or DM transfers and the overlay page.

A FIFO transfer is made up of at least one buffer transfer. If the transfer causes the pointers to wrap, a second buffer transfer is used. Rather than check for a wrap condition on each word transfer, a wrap condition is determined before transfers take place. If the total transfer will cause a pointer to wrap, the transfer is broken into two transfers—one at the bottom of the FIFO and another at the top.

The host must know the FIFO's location within the memory to interact with it. So, a tag holds the pointer and size information. Each FIFO has an associated tag that is placed in a predefined location within the mod870's memory. The host reads each tag before it performs transfers.

SUMMARY

To be successful in today's competitive RAS server market, you must build inexpensive, productive RAS servers. The two key ingredients in the recipe for success are a good RAS port IC choice and skillful integration of the RAS port IC in the RAS server. Make sure the RAS port is small, consumes little power, and has few pins. In addition, a programmable port that's easy to upgrade is helpful.

Also, apply the RAS port device the best possible way. Hardware and software integration must be approached properly for optimal results.

Take advantage of the port device's special serial and parallel I/O features. With hardware drivers, create a function for each transaction that the host can perform and build a generic function to abstract the hardware. You can use the API interface to create functions that support API mechanisms. ■

Shawn Arnold works at Analog Devices in the VoN Group of the RAS Product Line as an embedded software development engineer. Shawn held various engineering positions at ADI during the past 14 years, ranging from MCM and DSP Product Engineering to DSP and RAS Applications support. Shawn holds a BSEE and MSEE, both from Northeastern University. You may reach him at shawn.arnold@analog.com.

SOFTWARE

The software for this article is available for downloading from the *Circuit Cellar* web site.

SOURCES

ADSP-mod870-100 Processor
Analog Devices
(617) 329-4700
www.analogdevices.com

DS2151 TI Transiever
Dallas Semiconductor
(972) 371-4448
Fax: (972) 371-3715
www.dalsemi.com

MICRO SERIES

Op-Amp Specifications

Part 3: Input/Output Stages

Joe DiBartolomeo

Part 3 of 4

For Joe, putting this Microseries together has been a lot like working with op-amps—there's a lot more decisions to make than you would think. Just looking at the input and output stages reveals a host of options to consider.

While preparing this four-part Microseries, I discovered that writing about op-amps is similar to designing with them. When I dared to deviate from the world of the ideal op-amp, I found myself on a slippery slope. How much detail is too much, what do I leave out, how little is too little?

For example, the circuit in Figure 1a is a basic noninverting amplifier. Using the ideal op-amp model, you get the output voltage given in the equation. Many op-amp circuits can be designed using the ideal op-amp model and feedback equations.

If you need greater precision, take into account the op-amp specifications (errors) that I discussed in the first two parts of this article series (see Figure 1b). There are two equations in Figure 1b, the first is valid at a specific temperature and when there is a constant supply voltage. If there are thermal effects or power supply perturbations, you must use the second equation. But, note that I left things out. For example, I assume the op-amp differential gain is infinite. In addition, the equations in Figure 1 do not include op-amp or circuit noise and depict DC input errors only.

It is clear that your application will dictate which op-amp model to use and which op-amp specifications are important. What is often overlooked is that the external op-amp circuit can amplify or swamp out the affects of an op-amp specification.

The need to understand the total circuit will become clear shortly as we discuss op-amp input and output impedance along with the acceptable voltage ranges that allow proper op-amp operation.

Before getting started, I would like to finish Part 2's topics by presenting two circuits to test for op-amp bias currents, input offset voltage, CMRR, and PSRR (see the sidebar on page 67).

INPUT AND OUTPUT IMPEDANCE

Voltage feedback op-amps have both common mode and differential mode input impedance. Input impedance is specified in various ways in textbooks and manufacturers' databooks.

One popular way is to specify the differential impedance as the impedance between the two input terminals, represented by a capacitor and

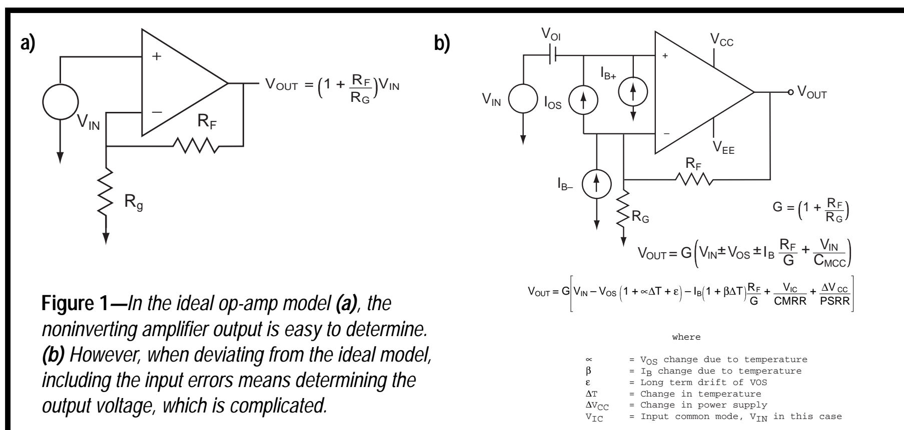


Figure 1—*In the ideal op-amp model **(a)**, the noninverting amplifier output is easy to determine. **(b)** However, when deviating from the ideal model, including the input errors means determining the output voltage, which is complicated.*

resistor in parallel. The common mode input impedance is specified as the impedance between each input terminal and ground, again represented by a capacitor parallel to a resistor. So, on the datasheets you see $10^{12} \,||\, 10$ in units of ohms and pF, respectively (see Figure 2a).

You also see the input impedance split, as shown in Figure 2b. The datasheet gives R_I and C_I, which represent the input resistance and capacitance between the terminals, with one terminal grounded. In Figure 2b, if you ground the positive terminal, $R_{IC} = R_D || R_N$ and $C_I = C_D || C_N$. C_{IC} and R_{IC} are specified for common mode signals. They are the input resistance and capacitance seen by a common mode signal with respect to ground.

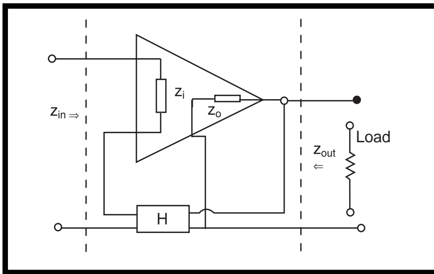


Figure 3—*The op-amp's input and output impedance is not what the load and source see. They see the input/output impedance of the circuit, which is the result of the op-amp impedance effected by the feedback network H.*

Connecting the two input terminals results in $C_D = C_P || C_N$ and $R_D = R_P || R_N$.

No matter how the input impedance is specified, the values range from 10^3 to 10^{12} ohms for resistance and 1 to 25 pF for capacitance. Input impedance is a nonlinear function of temperature and common mode voltage. For example, in FET devices, input common mode impedance is reduced by a factor of two for every 10°C rise in temperature. That's the same as bias currents, but in the opposite direction.

The output impedance of the op-amp is small—in the 10- to low-100-Ω range, mostly resistive. The op-amp output impedance is important in applications where a capacitive load is being driven and in power op-amp applications.

Usually the actual value of the op-amp's input or output impedance is not meaningful. As I discussed in Part 1, op-amps are rarely used in open-loop configuration. How the op-amp's impedance interacts with the external feedback circuit components to give the circuits overall input and output impedance is important.

Op-amp output impedance is in the tens to low thousands of ohms range, mostly resistive. As you will see, output impedance is almost always reduced by feedback. A couple of examples where output impedance is of concern are, driving capacitive loads or cables, or driving analog to digital converters (ADC) where the op-amp must charge the ADC input capacitance to within ½ LSB in a period of time that is less than 1 sampling interval. You'll see later how output impedance also affects the ideal loop gain

Op-amps are almost always used with feedback, normally negative. This feedback will either increase or decrease the input and output impedance seen by the source and load. Figure 3 points out that, in a standard op-amp circuit with feedback, the op-amps Z_I and Z_O are not what is presented to the load and source. What is presented is the result of the interaction between the op-amp's impedance and the external feedback circuit.

Taking a look at ideal amplifiers in feedback configurations, we can bundle them up into four basic topologies shown in Figure 4—series-series (SS), series- parallel (SP), parallel-series (PS), and parallel-parallel (PP).

The names stem from how the amplifier and feedback network are connected relative to the input and output signals. For example, in the SP configuration of Figure 4a, the amplifier input is connected in series with the feedback network while the amplifier output is connected in parallel with the feedback network. Understanding these configurations helps explain how the amplifiers impedance's interacts with the feedback network to give the overall input and output impedance.

If we go through some simple math for the SP configuration, you get:

$$V_{in} = V_{error} + BV_{out} \qquad [1]$$

$$V_{out} = AV_{error}$$

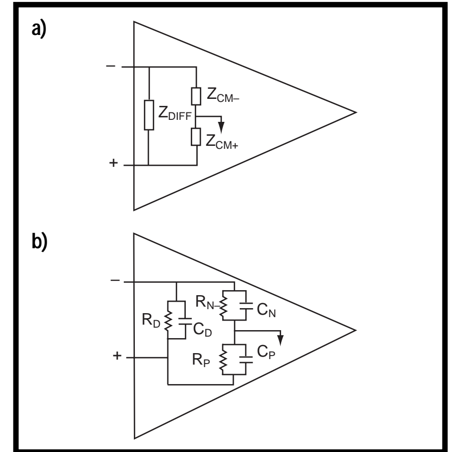


Figure 2—*Here are two common ways to represent op-amp input impedance.*

where *B* equals feedback network transfer function and *A* equals op-amp open-loop gain.

$$V_{in} = V_{error}(1 + AB) \qquad [2]$$

$$V_{in} = V_{error}(1 + AB) \qquad [3]$$

$$V_{error} = I_{in}Z_i \qquad [4]$$

$$V_{in} = I_{in}Z_i(1 + AB) \qquad [5]$$

$$Z_{IN} = \frac{V_{IN}}{I_{IN}} = (1 + AB)Z_i \qquad [6]$$

AB is greater than one, so the input impedance with feedback is greater than the amplifier's input impedance.

What happens to the output impedance? Here the feedback reduces the output impedance of the amplifier. A similar calculation to the one for input impedance would show that:

$$Z_{OUT} = \frac{Z_O}{(1 + AB)} \qquad [7]$$

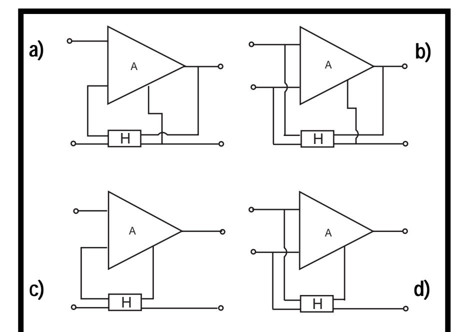


Figure 4—*Op-amps are almost always connected in a feedback network. The four ways of connecting are* ***(a)*** *series-parallel (SP),* ***(b)*** *parallel-parallel (PP),* ***(c)*** *series-series (SS), and* ***(d)*** *parallel-series (PS).*

A similar calculation for all four configurations can be performed (see Table 1). Op-amps are normally used in SP (noninverting) and PP (inverting) configurations.

Now, replace the ideal amplifier with an op-amp. Again, let's look at the three basic op-amp circuits—inverting, noninverting, and follower. We will look at both the ideal case and non-ideal case.

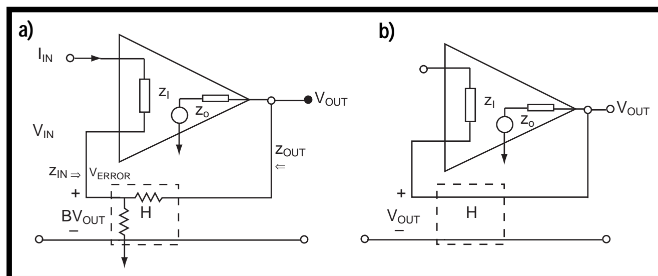


Figure 5—*In an op-amp circuit connected with negative feedback to produce a voltage amplifier, noninverting amp* ***(a)****, the feedback will increase the input impedance seen by the source and will decrease the output impedance presented to the load. The circuit also provides stable gain. These features make the noninverting amplifier suitable for voltage amplification.* ***(b)*** *If the gain is set to one, you have a voltage buffer.*

Starting with the noninverting amplifier of figure 5a, you see that there is a series parallel configuration. Here the input control variable is a voltage and the controlled output variable is also a voltage (this is a voltage amplifier).

Ideally what you have is the standard SP configuration that will yield a circuit input impedance as

Measurement of Bias Currents, Input Offset Voltage, CMRR, and PSRR

For years, each time I encountered a circuit to measure op-amp specifications, I was curious about its effectiveness. But, I never acted on my curiosity, assuming the results from these circuits would never match the manufacturer's specifications. Also, for most of my engineering career, I have had the luxury (curse) of working with either laser or radar systems. These systems are good at pointing out op-amp imperfections.

So, with this Microseries as an excuse, I built several circuits I found in various app notes, seminar notes, and textbooks (see Figures i and ii).

In Figure i, V_{IO} will appear across R and produce a current of V_{IO}/R, which will flow through R_F and produce an output voltage $V = V_{IO}/R \times R_F$. So, the value read with a voltmeter at V is approximately 1000 V_{IO}.

The bias current of the AUT can be ignored because V_{IO}/R is much larger then the bias current. Also, the effects of the buffer amp can be ignored because it must be a high-gain, low-input offset voltage and low bias current device.

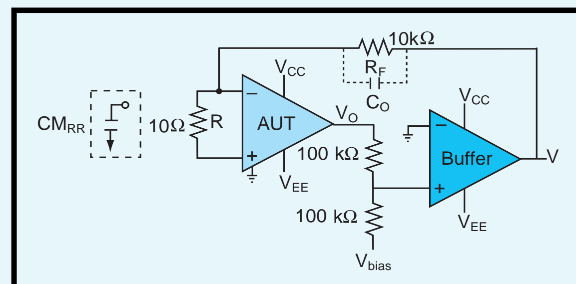


Figure i—*To measure input offset voltage (V_{IO}), set V_{BIAS} = 0. This puts V_O at zero, which is how V_{IO} is defined.*

To calculate PSRR, find V_{IO}. Then change V_{CC}/V_{EE} and again find V_{IO}:

$$PSRR = \frac{\Delta V_{CC}}{\Delta V_{IO}}$$

The greater the change in supply voltage, the easier it will be to measure the change in V_{IO}. Remember to change the supply rails equally in opposite directions to avoid a common mode change.

To measure CMRR, use the circuit to test for V_{IO} and apply a common mode voltage to the positive terminal, as shown in Figure i. Repeat for a different common mode voltage. You then get the change in V_{IO} for a change in common mode voltage:

$$CMRR = \frac{\Delta V_{CM}}{\Delta V_{IO}}$$

After you have V_{IO}, use the circuit in Figure ii to find the bias currents.

I tested several op-amps with various values of bias currents, offset voltage, CMRR, and PSRR. The tests yielded no surprises. Op-amps with the lowest datasheet specs also came up with the lowest test results, although I did not match the manufacturer specifications.

There are a few important things to take note of. As mentioned, the buffer amp must have high-gain, low-input offset voltage and low bias currents. Change the buffer amp and you will likely see changes in the output for the same AUT.

The resistor values may have to be played with, depending on the AUT. Also, if the circuit oscillates use a capacitor across the feedback resistor. These circuits give you a "feel" for the op-amp specs and are useful for those just starting out with op-amps.

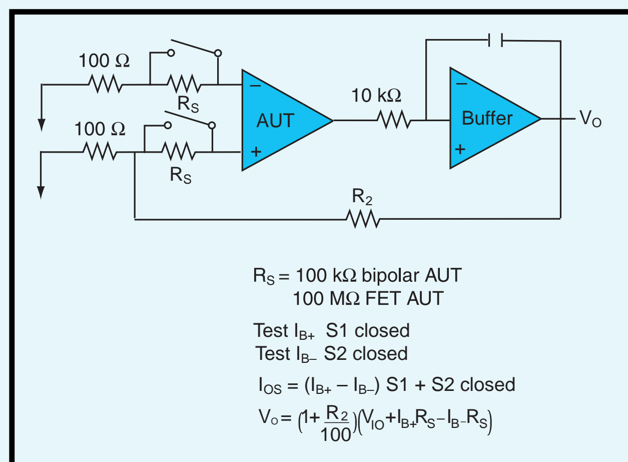


Figure ii—*You can use this circuit to find the bias currents.*

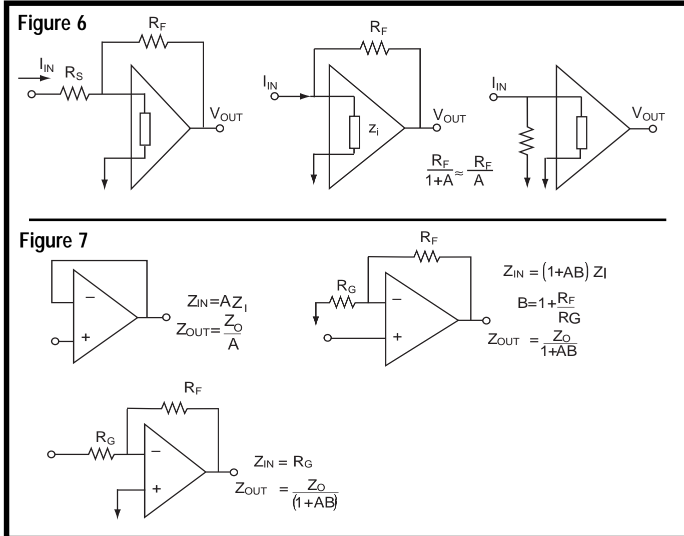


Figure 6—*The input impedance is R_S, whereas in* ***(b)****, the feedback resistor is Millerized to give the input impedance.*

Figure 7—*Here's the graphic version of Table 1 summarizing the affects of feedback topologies on the input and output impedance.*

given in equation 6. However, the circuit's actual input resistance is:

$$R_{IN} = R_D (1 + AB) \parallel R_C \qquad [9]$$

Where R_D and R_C are the resistive portions of Z_{DIFF} and Z_{CM} of Figure 2.

Here you can see that the differential input resistance is increased by feedback. At high gains, the op-amp's common mode resistance sets the resistance upper limit, as is also the case with a voltage follower.

The output resistance of the noninverting configuration is given by:

$$R_{OUT} = R_O/(1 + AB) \qquad [10]$$

as predicted by the general ideal case. But, note that the output resistance of the op-amp forms a voltage divider with the load and feedback resistors. The net affect is to reduce the open-loop gain—the larger the op-amp's output impedance, the greater the reduction in loop gain. As a rule of thumb, if the op-amp output resistance is one-tenth or less the value of the output resistance then its affects can be ignored.

In the inverting configuration ideally the input resistance is equal to the value of the series resistance, R_S (see Figure 6). You can see this if you take the case with no series resistor than the input resistance is the combination of R_D and the Millerized feedback resistor, R_F, so:

$$R_{IN} = (R_D \parallel R_F)/(1+AB)$$

Because R_D is generally much larger than R_F and AB is generally much larger than 1, you can approximate the equations as $R_{IN} = R_F/AB$, which is small. These returns use an input resistance that is equal to the series resistor, R_S.

The output resistance of the inverting configuration is reduced by feedback as was noninverting configuration, seen in equation 10.

Op-amp input resistance, both in inverting and non-inverting configuration, has an affect on loop gain because it is parallel with the series resistance. The approximation for the inverting case (see below) points this characteristic out:

$$B = [(R_S \parallel R_D)]/[(R_S \parallel R_D) + R_F]$$

if $R_D >>> R_S$, you get the familiar

$$B = R_S/(R_S + R_F).$$

If you were dealing with the non-inverting configuration this equation would be much more complex, but the results would besimilar.

Now, you see that the feedback network plays a dominant role in determining circuit impedance. But recall, feedback plays a dominant role in virtually all aspects of the op-amp circuit, from stabilizing gain and reducing distortion to increasing bandwidth. You saw last month that, in the case of the inverting op-amp circuit, you could ignore the effects of CMRR. However, in the noninverting configuration the effects of CMRR were amplified by the gain of the circuit.

Figure 7 summarizes the effects of feedback on the amplifier's circuits input and output impedance for most scenarios.

INPUT COMMON MODE VOLTAGE RANGE

The op-amp operates properly (linearly) in the input common mode voltage range, V_{icr}. Input common mode range, like many other op-amp specifications, can be explained by looking at the differential pair (see Figure 8a).

The biasing of the current source and current mirror results in an overhead voltage. This voltage puts a limit on the maximum input voltage. The transistors wouldn't be properly biased if any input voltage were higher or lower. Therefore, you must restrict the input voltage range to:

$$V_{rail} - (+)V_{overhead}$$

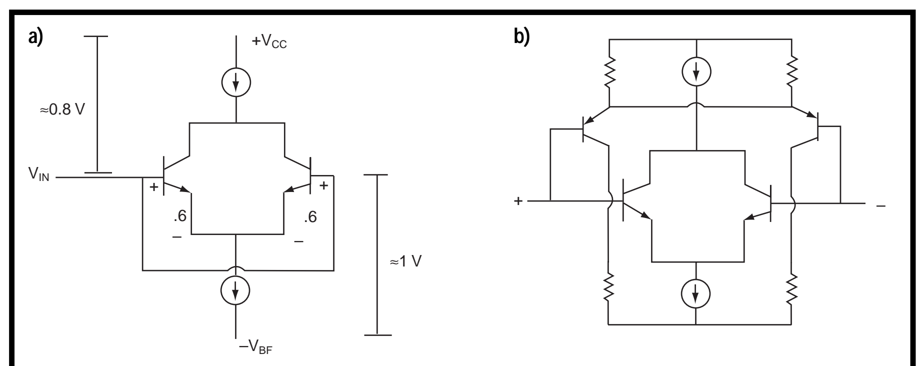


Figure 8a—*The voltage required to bias the current source and current mirror in the op-amps input stage is similar to an overhead voltage. This overhead means that the input common mode voltage can never equal either rail.* **b**—*Rail-to-rail input op-amps have complementary NPN and PNP differential pairs. This ensures that they can work with a common mode voltage equal to either rail because one pair is always active.*

The overhead voltage is different, depending on whether you are dealing with positive or negative rails, or the inverting or noninverting input (see Figure 8a).

For example, overheads are not critical for ±15 V rails, the input common mode range would be greater than ±10 V. However, if a single supply of 5 V and ground ran this op-amp, the common mode range would be less than 3 V. A rail-to-rail input op-amp's input structure is designed to extend the range as near to the rails as possible.

There are rail-to-rail input op-amps that include one or both supply rails. For example, op-amps with complementary NPN and PNP differential pairs can work with a common mode voltage equal to either rail because one pair remains active (see Figure 8b).

Exceeding the input range could cause an inverting circuit to become a noninverting circuit, or the inputs could be destroyed.

DIFFERENTIAL INPUT RANGE

There is a limit to the amount of differential voltage that can be applied across the op-amp's input terminals. This voltage is equal to or greater than the base emitter reverse breakdown voltage of one transistor plus V_{be} of the other transistor. Any excess voltage will damage the op-amp (see Figure 9a).

However, because the op-amp normally is used with feedback (i.e., both inputs at nearly the same values), usually differential input voltage range isn't a design issue.

Feedback topology	Transfer function	Z_{IN}	Z_{OUT}
SP	V_{OUT}/V_{IN}	Increase	Decrease
PP	V_{OUT}/I_{IN}	Decrease	Decrease
PS	I_{OUT}/I_{IN}	Decrease	Increase
SS	I_{OUT}/V_{IN}	Increase	Increase

Table 1—*Here is a summary of the effects of feedback topologies on an amplifier circuit's input and output impedance.*

OUTPUT VOLTAGE SWING

The best way to understand the output is to look at its structure. Figure 9b shows a common output stage of the emitter follower with complementary transistors, often called the class B push-pull amp. The name comes from the mode of operation, when V_I is positive, the NPN transistor is on and supplies the load current, while the PNP transistor is off. When V_I is negative, the PNP is on and supplies the load current while the NPN transistor is off. The diodes in Figure 9b greatly reduce the crossover distortion, which is caused by the transistor's V_{be}. The small resistors R1 and R2 help stabilize the quiescent current and improve small signal gain linearity.

Figure 9b clearly demonstrates that the emitter follower cannot drive the output to either rail. The output can drive:

$$V+ <= +V_{CC} - V_{R1} - V_{BEQ1} - V_{FAPQ1}$$

and

$$V- >= -V_{EE} + V_{R2} + V_{BEQ2} + V_{SAPQ2}$$

The maximum output voltage $\pm V_o$ is the maximum positive or negative peak output voltage that can be obtained without the output voltage clipping when quiescent DC output voltage is zero. This voltage depends on the load. The greater the output current, more is dropped across R1 and R2.

Rail-to-rail output op-amps use common emitter or common base rather than the emitter follower structure to extend the output voltage swing towards the rails. The output swing is limited only by the saturation voltage (bipolar) on resistance (CMOS) of the output transistors.

Now, you have looked at common op-amp specifications and how they affect circuit performance. Remember that these specifications are effected significantly by the input and output topologies.

Next month, I'll look at AC op-amp specifications and the trade-off when selecting an op-amp for a particular application. ◼

Joe DiBartolomeo, P.Eng., has more than 15 years of engineering experience. He is currently employed by Texas Instruments as an analog field engineer. You may reach him at j-dibartolomeo@ti.com.

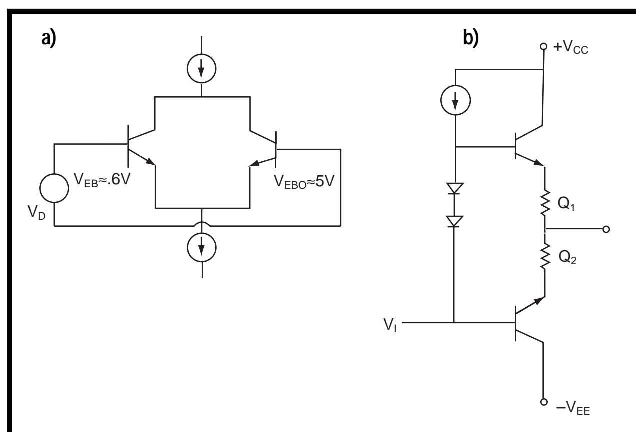


Figure 9a—*Any voltage that is equal to or greater than one base emitter, reverse breakdown voltage, plus V_{be} causes excess current flow in the inputs and may destroy the op-amp.* **b**—*This is a common op-amp output stage. The emitter follower with complementary transistors is often called the class B push-pull amp. Because this configuration won't allow the output to reach either rail, rail-to-rail op-amps use common emitter or base configurations.*

REFERENCES

Analog Devices, Linear Design Seminar, Analog Devices, 1995.

Coughlin and Driscoll, *Operational Amplifiers and Linear Integrated Circuits*, Prentice Hall, 1977.

P. Horowitz and W. Hill, *Art of Electronics*, Cambridge Univeristy Press, NY, 1990.

J. Karki, "Understanding Operational Amplifier Specifications", Texas Instruments, White Paper, 1998.

A.B. Malvino, *Electronic Principles*, McGraw-Hill, NY, 1979.

Miller, *Microelectronic*, McGraw-Hill, NY, 1979.

Texas Instruments, *TI Analog Seminar Handbook*, Texas Instruments, White Paper, 1999.

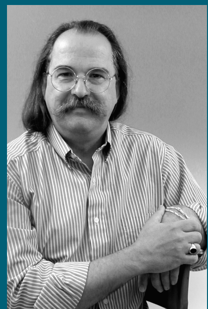
M.E. van Valkenburg, *Analog Filter Design*, Holt, Reinhart and Winston, 1982.

The Chips are Alive with the Sound of Music

FROM THE BENCH

Jeff Bachiochi

Imitating the Dead Melody IC



The Von Trapp family may have made some great music, but try fitting them all into a birth-day card and you'll see why Holtek's Melody Generator ICs were such a hit.

used to make an annual pilgrimage to New Jersey for the Trenton Computer Fair. It wasn't like today's computer fairs where John Q Public roams the aisles for the cheapest Taiwanese clones. Most of the products were made by hand for the elite who knew what a computer was before Macintosh and Microsoft.

Outside, hackers (back when a hacker was a good guy) and hams sold the electronic stuff that had been collecting dust since they bought it the previous year. I picked up the first singing birthday card I ever saw at a fair. "Wow," I thought, "How did they get all that into a card?" I plunked down five bucks and ogled at the little bits inside the folded paper greeting.

I had to search high and low for an old Digikey catalog. I learned not to throw out an old databook when the new one arrives, but didn't think I'd need an old Digikey catalog. However, I knew I had seen these things in there, the Holtek Melody Generator ICs, a TO-92 device with connections for a battery and a piezo output device (see Figure 1). They cost less than $1 and were available in a selection of melodies. But, Digikey no longer lists them. A trip to Holtek's web site confirmed my fears that they were discontinued.

PERSONAL MELODY

I couldn't let this discourage me. I wanted to make a special card for my wife this year and I planned to use one of Holtek's parts for it. I ended up turning a simple gesture into a full-fledged project (see Photo 1).To make a similar device requires a programmable microprocessor. A small 8-pin SMD device would be the perfect size for a Holtek replacement. The only manufacturers of 8-pin micros that I know of are Atmel and Microchip. Microchip's PIC12C508 does not have interrupts, so it can't easily output frequencies with stability amidst the other processing that's necessary. This means the smallest workable PIC is the PIC12C671. Atmel's AVR series 8-pin micros have timer interrupts available, so the ATtiny10, which is its bottom-end device, is adequate.

TOS

In the synthesizer industry, TOS means top octave synthesizer, which was developed in the '70s for electronic organs. A 1- or 2-MHz input would be internally divided to produce 12 equally spaced frequency outputs. The 12 semitones comprise the musical scale, an octave. An octave is the interval between two notes where one frequency is twice or half of the other. When an octave is produced, dividing a note's frequency by two derives the remaining notes in lower octaves. Hence, notes are based on the single clock's input frequency.

Because TOSs are hardware devices, no overhead was necessary. Doing this with a processor requires hardware PWMs or an execution speed

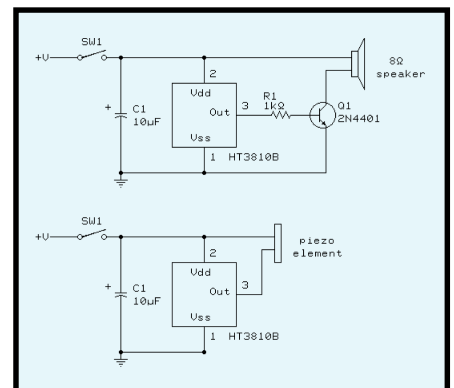


Figure 1—*Holtek melody generators were popular, but they have since been discontinued.*

that supports software PWM. Hardware PWMs are not available on these low-end micros. Execution speed of both of these is 1 µs using their internal 4-MHz oscillators. Software divide-by-N counters can be coded using an interrupt routine. Look at Figure 2 to see how this is accomplished.

At least two divide-by-N counters are part of the timer overflow interrupt, there is an accurate time base tick. Upon N counts or ticks, an action is performed. Here, two actions are necessary. The first, frequency creation, is done using an 8-bit count and by complementing an output bit each time the divide-by-N reaches zero. Two transitions of this bit equal one cycle of output frequency (period = two divide-by-N counts).

The second action concerns duration, which requires a 16-bit count. This is the duration of the fastest note, in this case I'm using the eighth note (one beat in x/8 time). The longest path through this routine must include reloading the "N" each time a divide-by-N counter reaches zero for a single note (frequency counter) and a 23-µs duration (beat counter). Although I could reload timer0 with any number greater than the maximum interrupt time, I used 64 µs to give the rest of the code enough time to execute between interrupts.

FIRST COMPARISON

Both chips execute an instruction in 1 µs. Although Microchip's instruction set is simpler, many operations require multiple instructions because data must move through the W register. One of Microchip's powerful instructions is the `decfsz` command, which decrements, tests for zero, and determines a branch in a single instruction and one instruction cycle (see Listing 1). It takes two cycles if it must change the program counter.

My interrupt timer routines require the same number of maximum instruction cycles for either processor. Frequency-generated notes are based on the 64-µs timer interrupt routine. Each note's duration is based on a tic or beat of 1000h interrupts (~0.25 s). The rest of the code is a small loop and two main lookup tables. The first lookup table is an 8-bit code containing note, duration, enable, and EOF information. See Listing 2 for the bit information arrangement.

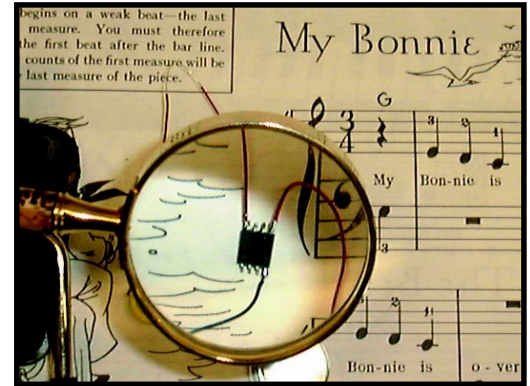


Photo 1—*A complete melody generator consists of an SMD 8-pin micro, coin battery, and piezo device.*

Listing 1—*The `decfsz` command determines a branch in a single instruction and one instruction cycle.*

```
  NOTDONE:     DECFSZ            TEMP  ; 1, 2ifz - decrement and skip if zero
  GOTO         NOTDONE           ; 2
  DONE:        (minimum of two execution cycles)
//Atmel's equivalent code requires three instructions, dec, tst, and breq and
four to five instruction cycles.
  NOTDONE:     DEC               TEMP     ; 1
               TST               TEMP     ; 1
               BREQ              DONE     ; 1, 2ifz - branch if zero
               RJMP              NOTDONE  ; 2
   DONE:                         (minimum of four execution cycles)
//Of course, you would likely use the opposite branch instruction and save a
cycle.
      NOTDONE: DEC               TEMP        ; 1
               TST               TEMP        ; 1
               BRNE              NOTDONE  ; 1,2if<>z - branch if not zero
  DONE:        (minimum of 3 execution cycles)
//If you had a spare register that was cleared, you could use this (TEMPZ)
register to save another cycle.
  NOTDONE:     DEC               TEMP        ; 1
               CPSE              TEMP,TEMPZ  ; 1 - compare and skip if equal
               RJMP                      NOTDONE     ; 2
  DONE:                          (minimum of two execution cycles)
```

Listing 2—*Here's the bit information arrangement.*

```
    1xxxxxxx = Last note in list
    0xxxxxxx = More notes in list
    x1xxxxxx = Disable the frequency output (rest)
    x0xxxxxx = Enable the frequency output (note)
    xx11xxxx = Duration eight half beats (whole note)
    xx10xxxx = Duration four half beats (half note)
    xx01xxxx = Duration two half beats (quarter note)
    xx00xxxx = Duration one half beats (eighth note)
    xxxx0000 = Note table offset zero (note "G")
    …
    xxxx1111 = Note table offset 15 (note "F")
```

Here's how it begins. After initial housekeeping, `Table1offset` is cleared, and the first value of Table 1 is read. If the MSB is a one, you are at the last note in the tune, and the `Table1offset` is cleared, allowing the tune to repeat. Otherwise, `Table1offset` is incremented for the next table1 read. Now,you have the data indicating which note will be played and its duration.

Next, `Note1duration` is initialized to one, and if the LSB duration is one, `Note1duration` is shifted left. If the MSB duration is also one, `Note1 duration` is shifted left twice. This shifting sets the duration to one, two, four, or eight. This is used to count off the note's beats. Remember the second (16-bit) counter in the interrupt routine? That counter is responsible for the beat timing (of an eighth note). In this case, the beat is arbitrarily set to ~0.25 s (64 µs × 1000h). Now you've set up the number of tics the note should last, one for 0.25 s (the shortest note), two for 0.50 s, four for 1 s, or eight for ~2 s (the longest note).

The note number (LS4B) of the first table read is used as an offset in a second table. This table holds the `Note1reload` value. The first table doesn't need to know how to produce the note, just where it's located in the

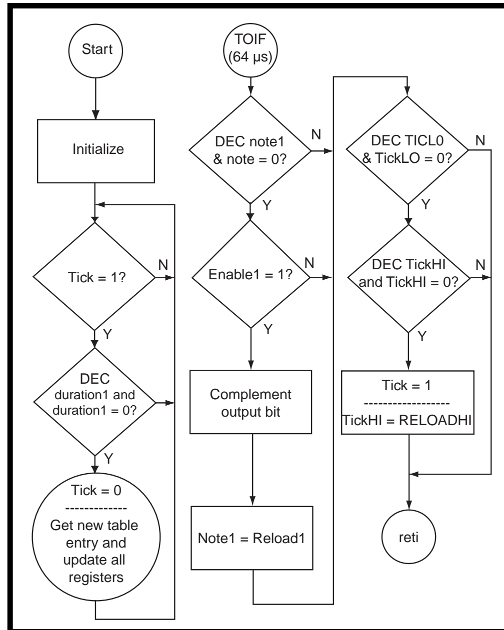


Figure 2—*The melody generator flowchart shows frequency and duration based on a single timer.*

crosses a page boundary. A smaller word means smaller die sizes, and that spells savings for the customer. However, special handling can be confusing when working with larger tables. Because Atmel uses two 8-bit registers (16-bits) for indirect addressing, table lookups are easier to manage.

Atmel states program memory in 8-bit bytes. Because instructions are in words, you must adjust the program size accordingly (divide by two). However, table values are stacked two to a word. Microchip states program memory by instruction width (word), such that an instruction requires one program word. But, each table value also requires a full word. There are pros and cons to each methodology, therefore it is important to know about each processor so you can choose the best one for your design.

second table. Thus, the second table can be transposed easily without worrying about the duration of the note. In fact, the notes don't have to be in a particular order and can span octaves.

`Note1reload` has a count that produces periods where an output bit is complemented every time the count reaches zero, which is the frequency output of your note. `Note1duration` has a count that counts off 0.25-s beats, which is the duration of your note. When the counters are set, the interrupt is enabled, and you're ready to begin.

The frequency output continues for the note's duration. When `Note1 duration` is zero (by 0.25-s tics), execution goes back to reading another table entry. The loop continues reading table entries until it finds an entry containing the MSB set, when it clears `Table1offset` and starts the tune again. Bit 6 (enable) indicates a noteless duration interval, called a rest. A one in this bit location disables the frequency output for the duration period.

SECOND COMPARISON

Many programs require table lookups. Microchip's low- and mid-range processors have a single 8-bit indirect register for table lookups. This requires special handling when the table gets larger than 255 entries or it

SOUNDS LIKE

Square waves don't sound smooth. They are irritating on their own, as proven by cell phones and beepers. However, piezo elements allow thin transducers where fidelity (at least the bandwidth is greater than the telephone) is not necessary. The devices' frequency range fits well into that limited bandwidth. The piezo element can be directly driven from the microcontroller I/O port's toggling bit. Although you could get more drive from a transistor-driven speaker, if the idea is to hide the circuitry within a greeting card, the speaker is out of the question (see Figure 3).

Here's how a simple tune translates into a table of entries. Middle C on a piano, and all of the notes within an octave above it, are noted by their capital letters. Notes one octave below middle C have a tick before them, and notes one octave above middle C are followed by a tick.

Notes in octaves further away have multiple ticks. With this notation, you can determine a note's exact location. Looking at a piece of music, I copied the musical notations and timing duration of each note into a

list of notes. Is it an eighth, quarter, half, or whole note? The song "My Bonnie (Lies Over the Ocean)" is in 3/4 time. This means there are three beats to the measure and a quarter note is equal to one beat (eighth note = half beat, half note = two beats, and whole note = four beats). Measures break the music into manageable pieces.

Look at Table 1's second column, which lists the notes and how many halves of a beat each note gets. I use halves of a beat because the fastest note I can handle is an eighth note (two to one beat or half beat each), even though there are none in this song. Notice that each group adds up to six (half beats), that's three beats to a measure. Don't worry about the first and last group, this is a partial listing of the song, they also total three beats. The last column shows the words to the song, so you can follow what I'm writing about.

The first column in Table 1 is a binary representation of the note and duration. This is the data the micro gets from the table. It also contains other important information. By equating some initial constants for all the possible notes, durations, rests, and last note indication, I could let the assembler calculate the data values for every entry in the table.

TEA FOR TWO

The interrupt routine requires less than the 64 µs that timer0 allows. If it didn't, no other work could be done. It has enough time, in fact, that the note generation code can be duplicated to allow for a second note. The second note would be output at the same time as the first, creating harmony. In addition, a second note requires a second note table. This isn't a prob-

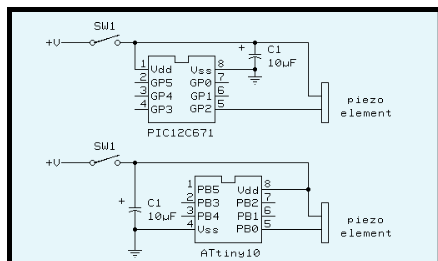


Figure 3—*An 8-bit micro makes a simple melody generator. Both Microchip and Atmel parts can be used.*

00010111	G-2	My	00010100	C-2	ver
00010010	E-2	Bon	00010110	'A-2	the
00010011	D-2	nie	00010111	'G-2	o
00010100	C-2	lies	00101001	'E-4	cean
00010011	D-2	o	00101001	'E-4	…

Table 1—*These groups contain notations for each measure that is to be played.*

lem, but it created unusual output when I entered one wrong timing bit. The first time through the tables, the notes were in time with each other. However, after the incorrectly notated note played, the two parts were out of sync. And, each time through, they grew further apart. This is where separating the table data into groups (measures) makes it easy to locate errors.

PIC AND CHOOSE

Both Microchip and Atmel's microprocessors can do the job. Comparable tools are available for Microchip and Atmel's 8-pin wonders. Microchip has a wide selection. Atmel's larger instruction set handles the total program space easier, despite confusing register restrictions. On the other hand, Microchip's smaller instruction set is easily memorized. Although its handling of tables is not as straightforward as Atmel's 16-bit pointer, it works well for small tables. Although the Atmel device handles table space efficiently, Microchip's instruction set provides more compact code.

The interrupt routine is critical in this project, and neither processor has an execution advantage over the other. So, there isn't any overwhelming reason to choose one over the other.

Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on Circuit Cellar*'s engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

SOURCES

ATtiny10/11/12
Atmel Corporation
(408) 441-0311 • Fax: (408) 436-4200
www.atmel.com

PIC12C671
Microchip Technologies Inc.
(480) 786-7200 • Fax: (480) 899-9210
www.microchip.com

On The Road Again

SILICON UPDATE

Tom Cantrell

Part 2: Taking Silicon for a Test Drive

What exactly is it that makes the car business tick? Tom's not sure, but he did take a look under the hood to see just how challenging it can be to service a car that's loaded with silicon.

buying a new vehicle is an amusing departure from the no-haggle price tag approach that characterizes most sales transactions. Imagine the same situation with, say, a loaf of bread. After eight hours at the checkout counter, envision your earnest sales consultant (aka, the good guy) repeatedly running to the back room to plead your case to the sales manager (aka, the bad guy).

Why is this odd game played with car purchases? I think the persistence of the let's-make-a-deal approach is a result of high prices and the fact that most people mistakenly think they're savvy negotiators.

Recently, I took the plunge and bought a new Chevy van. I put up an admirable fight (hint: bring your own bad guy, aka, your wife), but sincerely doubt the dealer lost money on the deal. Anyway, purchasing the new van is the reason I'm suddenly on this car computing kick. As you'll see, the collision of modern technology and the somewhat bizarre ways of the car business makes a trip under the hood of a modern car worthwhile.

HOOKING UP

Testimony to the fact that times have changed, when I ordered shop manuals for the van, I discovered almost 600 pages devoted to the subject of engine controls versus a mere 200 for engine mechanical.

Interest piqued, I poked around under the dash and found, as you will on any '96 or later vehicle, the Data Link Connector (DLC) shown in Figure 1, which serves as a gateway into the vehicle's LAN.

Note that out of the 16 pins, only the power, ground, and data line(s) (whether J1850, ISO, or CAN) are strictly specified (in SAE J1962). In GM vehicles, the single-wire version of J1850 uses pin 2. The rest of the pins are available for proprietary use by each manufacturer.

When you take your car for service, a scan tool gets plugged into the DLC. As you'll see, this allows the repair technician to quickly and easily get a good picture of what is, and what has been, going on under the hood.

Perusing the shop manuals further, I found that my van incorporates all manner of electronic sensors and subsystems that operate under the watchful direction of the Vehicle Control Module (VCM, see Photo 1).

According to the specs I found at Delphi (the spinout of formerly captive GM parts-supplier, Delco), my VCM incorporates a 16.7-MHz 32-bit

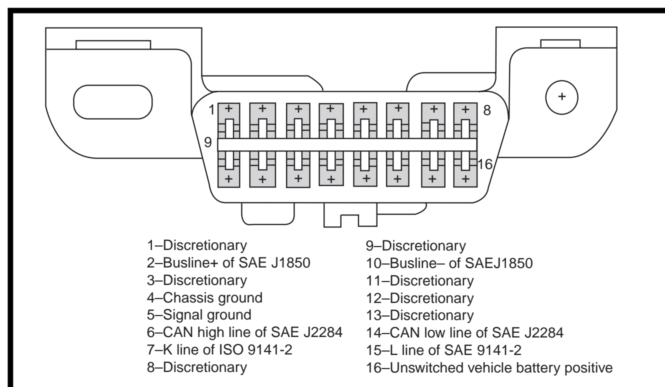


Figure 1—*Although the protocols (J1850, ISO, CAN) vary, at least the pin description and form factor of the Data Link Connector have been standardized. As a result, the trend is towards universal test equipment, such as scan tools that work with any car.*

microcontroller (MC-68332), a healthy half-megabyte of flash memory, and 6 KB of RAM.

With 160 I/O connections, the to-do list for the VCM is certainly full. Practically on a rev-by-rev basis (i.e., hundreds of hertz), the VCM deals with the myriad of sensors and actuators responsible for low-level control of the powertrain—spark timing, fuel/air mixture, transmission shift points, and so on. Furthermore, the VCM continuously runs reality checks to detect component failures and compensate as best it can (i.e., limp home).

```
40 82 48 6B 10 41 0C 0A E0 23 00 00 00 9F
40 82 48 6B 10 41 0C 0A D9 9C 00 00 00 11
40 82 48 6B 10 41 0C 0A FE 48 00 00 00 E2

40 82 A8 FF 29 03 F0 00 00 00 00 00 00 45

40 82 48 6B 10 41 0C 0A DC F5 00 00 00 6D
40 82 48 6B 10 41 0C 0A E6 6D 00 00 00 EF
40 82 48 6B 10 41 0C 0A EE 85 00 00 00 0F
40 82 48 6B 10 41 0C 0A F3 C9 00 00 00 58

40 82 E8 FF 10 03 B3 00 00 00 00 00 00 2F

40 82 48 6B 10 41 0C 0A F9 1B 00 00 00 B0
40 82 48 6B 10 41 0C 0A F4 9A 00 00 00 2A
40 82 48 6B 10 41 0C 0A E9 D6 00 00 00 5B
```

Figure 2—*The first few captured messages illustrate a valid RPM response. For example, in the first line, 0AE0 hex represents 4× the RPM (i.e., RPM = 2784/4 = 696). The purpose of the following byte (e.g., 23 hex in the first line), not to mention the intermittent mystery messages (third byte not equal 48), remains to be determined.*

On a higher plane, the VCM coordinates activity across the powertrain as a whole, notably getting the engine, transmission, and brakes to work as a team. For example, as you travel over hill and dale, the VCM will update transmission shift characteristics depending on the circumstances, such as whether you're towing a trailer at the time.

MIL SPEC

One fine morning I started the new van and left it idling while I ran back in the house. Upon jumping back behind the wheel, the "Service Engine Soon" message was glaring at me from the infamous malfunction indicator lamp.

With only about 500 miles on the clock, I wasn't exactly a happy camper. According to the owners manual, driving the vehicle with the MIL on could lead to costly repairs that may not be covered by my warranty.

That's just perfect. Though the van seemed to be running fine, the ominous wording left little choice but to make a trip to the dealer.

Fortunately, it turned out to be a false alarm. The repair technician hooked up a scan tool to the DLC and extracted Diagnostic Trouble Code (DTC) P0141, meaning the VCM felt that one of the oxygen sensors wasn't heating up properly. I don't know if the technician performed the intricate diagnosis of the oxygen sensor called for in the manual or just cleared the DTC (extinguishing the MIL), but the problem hasn't popped up since.

Nevertheless, the episode prompted me to investigate further. Not to impugn anyone's integrity, but forewarned is forearmed. When I go in for repairs, I like to be clued in and prepared to some extent.

I started cruising the Internet and found myself immersed in the regulatory depths of the Environmental Protection Agency (EPA) and the California Air Resources Board (CARB). It turns out that one of the VCM's most important roles is that of smog buster (i.e., continuously monitoring and tweaking operation to minimize air pollution). If the VCM thinks something isn't kosher, it turns on the MIL.

Of course, everyone supports clean air, but be assured the advocacy is not all altruism on the part of car manufacturers. EPA and CARB may walk softly, but they carry a big stick.

In September '98, for example, CARB ordered the recall of 330,000 Toyota and Lexus cars to replace onboard computers that failed to detect gas vapor leaks under normal driving conditions. At a cool $250 each, that adds up to more than $80 million. Rather a strong incentive to get it right next time.

Indeed, vapor leaks are considered a major pollution no-no, so modern vehicles have sophisticated EVAP (evaporative) systems that recycle fuel vapors. It's up to the onboard computer to ensure that the fuel system doesn't leak, and that's why something so simple as not tightening the gas cap can turn on the MIL.

Another suspect to watch out for is misfire, incomplete combustion that not only pumps raw gas out the tailpipe, but can damage the catalytic converter. The VCM goes to great effort to detect misfire by statistically sampling variations in crankshaft speed and using camshaft position to isolate the problem to a single cylinder. To complicate matters, something as common as bouncing over a pothole can feedback through the drivetrain and mimic misfire. So, in the most advanced implementations, misfire detection is qualified by a "rough road" input from the brake or suspension module.

Even when all is well, a lot of pollution occurs in the first few minutes after a cold start. The challenge for the VCM is to achieve reliable starts and smooth idling without just throwing extra gas at the problem (as with yesteryears' choke).

That's why oxygen sensors have heaters. They need to be up to proper operating temperature before the VCM can enter the closed loop mode that continuously varies the fuel/air mix to achieve maximum performance and fuel economy with minimum smog.

OH BOY, DATA

Thanks to the regulatory might of the EPA and CARB, the name of the game when it comes to onboard computing is OBD II, the second-generation on-board diagnostics mandated by law since model year '96.

While wandering around on the Internet, I perceived what appeared to be a tug-of-war between the various

players who have high stakes on the table. The car companies seem biased toward keeping the inner workings a deep, dark secret. Meanwhile, the aftermarket repair shops are alarmed at the prospect of cars that can only be fixed by an authorized dealer.

Striking a balance between the two are the regulators. They want the procedures and tools needed to fix a smoker widely accessible, yet understand it's not reasonable to expect manufacturers to disclose all. For example, the VCM not only controls the engine, transmission, and such, but also plays a role in theft prevention (e.g., disables starter in absence of a valid key signal).

Frankly, none of them seem too eager to let the lowly consumer have a clue. In particular, the regulators look askance at the emergence of power chips and such, going so far as to demand sophisticated countermeasures to prevent homebrew hot-rodding (i.e., reprogramming) of VCMs.

OBD II is the compromise solution. It openly standardizes (in SAE J1979) access via the DLC to the real-time (RPM, temp, vacuum, etc.) and historic freeze-frame information deemed necessary to correct emissions-related problems. It also defines the mechanism for retrieving and clearing DTCs (i.e., find out why the MIL went on and turn it off).

SCAN MAN

With the emergence of standards, as a trip to your local auto parts store will confirm, the average shade tree mechanic now has access to scan tools that were formerly the domain of dealers only. These are typically hand-held units that cost $200 or so, and get by with a keypad and small LED display or LCD screen.

There are also PC-based solutions on the market that offer pretty screens, hard-disk data logging, printed reports, and so on (see Photo 2). Generally, they use a converter dongle that connects the J1850 DLC to the PC's serial port. Because the price is about the same as the hand-held units, these are likely to have more appeal for computer literate types. I discovered some interesting units available from Baum Tools, B&B Electronics, and Ease Simulations.

Whether hand-held or PC-based, these units are generally intended for hood-up inspection, maintenance, and repair. Indeed, as one supplier's documentation notes, there's no easier way to end up in a ditch than trying to drive while you're fiddling around with one of these gadgets.

I was intrigued by the possible on-the-road applications. All I needed was a simple J1850-to-RS-232 adapter with enough brain power to get online.

Fortunately, I stumbled across an outfit called Multiplex Engineering, which offers such a gadget for less than $100 (see Photo 3). It was time to try my hand at a little high-horse-power hacking.

ACCESSORIES

Though Multiplex Engineering will take an order for a single unit, be advised that they primarily deal with OEMs, rather than end users. Thus, documentation is rather sparse and deals mainly with the basics of estab-

Listing 1—*Notice that mystery messages (third byte not equal 48 hex) are simply ignored.*

```
PROGRAM J1850Tach
INTEGER rpm,i,j
INTEGER response(14)
STRING rpm_str
CONST leds=~$c038~
BEGIN

/* Mux interface msg to request RPM */
DATA  $20,2,5,$68,$6a,$f1,1,$c,0,0,0,0,0,0,$d7

OUT 0,$74     /* '180 ASCI0 19.2K 8N1 */
OUT 2,$20
OUT 4,0

FOR i=0 TO 7
  OUT leds+i,ASC(" ")
NEXT i
OUT leds,ASC("R")
OUT leds+1,ASC("p")
OUT leds+2,ASC("m")

loop:
/* Send request RPM msg */
  FOR i=1 TO 15
    READ j
    DO
    UNTIL BAND(INP(4),2)<>0
    OUT 6,j
  NEXT i

/* Receive RPM response msg */
  FOR i=1 TO 14
    DO
    UNTIL BAND(INP(4),$80)<>0
    response(i)=INP(8)
  NEXT i

/* Throw away mystery msgs */
  IF response(3)<>$48 THEN GOTO loop

/* Convert RPM to 3 or 4 digit string, display on LEDs */
  rpm = ((response(8)*256)+response(9))/4
  rpm_str = STR$(rpm)
  IF LEN(rpm_str)=10 THEN rpm_str=MID$(rpm_str,1,4)
  ELSE rpm_str=CONCAT$(" ",MID$(rpm_str,1,3))
  FOR i=0 TO 3
    OUT leds+4+i,ASC(MID$(rpm_str,i+1,1))
  NEXT i
  WAIT 6  /* Spec requires 100ms min between msgs */

GOTO loop
```

lishing J1850 communications. Just as an Ethernet chip datasheet doesn't teach you about TCP/IP, the documentation tells you a lot about how to talk over J1850, but little about what to say.

Thus, the mux interface by itself isn't a substitute for the higher priced turnkey PC-based scan tools. However, if you aren't afraid to get your hands dirty, it is a good alternative to rolling your own (e.g., starting from scratch with one of the J1850 chips described last month) for applications where a PC is overkill.

With that in mind, I decided to have a go at it and see if I could get a BASIC SBC to have a meaningful J1850 dialog with my van.

The first step was to establish basic communication on the RS-232 side. This sounds simple enough, but RS-232 hook-ups invariably involve more hassles than they should, and this time was no exception.

No problem with the 19.2-KB, 8–N–1 part, but I did have to ponder a bit about the handshaking lines. The unit is wisely optoisolated to keep load dumps and the like at bay.

To accomplish this, the RS-232 port is powered from the host via the DTR and RTS lines. So, whatever you hook up must connect these lines and be capable of driving them to opposite RS-232 polarity, which means you need three distinct transmitters (TX, DTR, RTS) along with a receiver (RX). The SBC's MAX-232 chip has only two transmitters, so there wasn't a way to cut and jump around the problem.

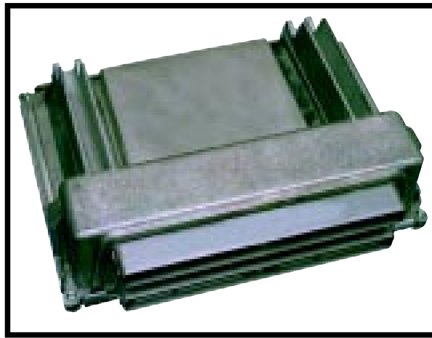


Photo 1—*With 16.67 MHz 68K, a half-megabyte of flash memory, and 160 I/Os, the Delphi Vehicle Control Module is where silicon meets the road.*

Whenever I need to make an RS-232 connection, I reach into my stockpile of MAX-235 chips. Yes, the 28-pin wide DIP is bulky, but it's also much handier for prototyping than a tiny surface-mount package. Otherwise, with five transmitters and receivers, a single 5-V supply, and no external components (except for a 1.0-µF power supply bypass capacitor), the MAX-235 is definitely the Cadillac of RS-232 chips.

Because I'm going upscale with RS-232, why not go for the gusto with the display as well? Once again, reaching back into the trusty group of gadgets I've used before (*Circuit Cellar* 31, "Smart LEDs: The Hard Way, the Soft Way, and the Right Way"), I came up with the HP DSP-2501, 8-character bit-mapped LED display.

It isn't cheap, and the parallel interface demands a lot of I/O lines, but the display quality is unsurpassed compared to the inferior LCD. Is it sunlight readable, you ask. Well, if it weren't for the software dimming

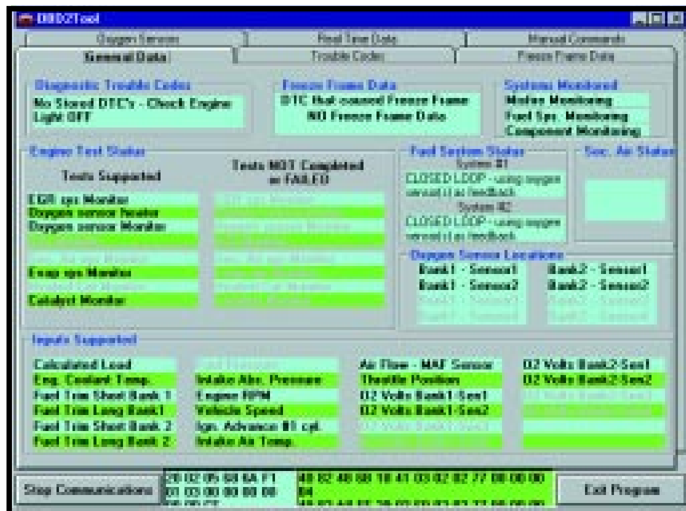


Photo 2—*PC-based scan tools use software (in this case, OBD2-TOOL from Baum Tools), running on a PC to offer more flexibility and features at a lower cost than hand-held units.*

feature, you'd practically need sunglasses to look at it.

Not surprisingly, the HP unit, at full throttle with 280 individual LEDs (eight characters, each 5 × 7), calls for a high-octane power supply. In fact, the datasheet has cautionary language about limiting the total power (i.e., number of LEDs on at once), lest things get a little too hot to handle.

Fortunately, there's no shortage of engine-on power in automotive apps, considering the typical cigarette lighter outlet can deliver 100 W or so. For the few watts of 5 V or more required by the SBC and display, I might have been able to get by with a plain linear regulator, but it would have been iffy. Even under normal conditions (about 14 V, according to the van's voltmeter), the regulator would get hot, not to mention coping with worst-case spikes. So, I took the easy way out with a fully-loaded 15-W DC/DC converter featuring 9- to 18-V input and overvoltage, thermal, and reverse polarity protection.

TACH TIME

Time to buckle up and hit it. The Multiplex Engineering mux interface puts a wrapper (i.e., destination, command, number of J1850 bytes, checksum) around the raw J1850 messages. One nice feature is that your software only needs to deal with a simple checksum because the mux interface handles the J1850 CRC. Another simplifier is fixed-length packets, with unused bytes zeroed.

Care for an example? To find out the current, RPM requires sending the following J1850 packet:

68, 6A, F1, 01, 0C, 8B

The first three bytes are the J1850 header, *01* is the code for mandated OBD II diagnostics, *0C* specifies RPM, and *8B* is the J1850 CRC.

So, what you need to send to the mux interface is:

20, 02, 05, 68, 6A, F1, 01, 0C, 00, 00, 00, 00, 00, 00, D7

The first byte (*20*) is an address associated with a particular mux interface. The next (*02*) is a command to send a J1850 VPW message. The third byte is a count of the J1850 packet length, not counting the CRC. The next five bytes are the J1850 packet (minus CRC) followed by six bytes of zero pad (recall the maximum J1850 packet length is 11 bytes). The fifteenth and last byte is the checksum of all bytes except the first.

Coming back on J1850, you expect to see something like the following, where HH and LL are the high and low bytes of the RPM:

48, 6B, 41, 01, HH, LL, CRC

Oh, by the way, don't forget it's actually RPM × 4, or you might end up with gray hair, like someone I know.

In turn, the mux interface will return the following, where CHK is the checksum depending on the RPM:

40, 82, 48, 6b, 41, 01, HH, LL, 00,00, 00, 00, 00, 00, 00, CHK

Refer to Listing 1 to see the BASIC embodiment of all this. Without further ado, as you can see in Photo 4, I was successful in my quest to make the J1850 connection.

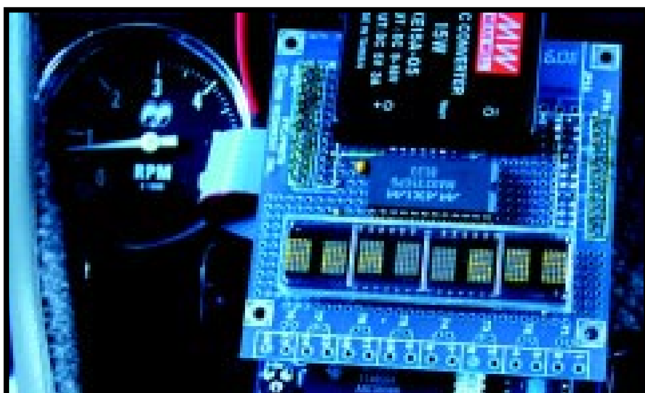


Photo 4—*The old and new. Though seemingly analog, the dial gauges in the instrument panel are driven digitally. Even the radio is connected to the VCM, so the faster you go, the louder it gets.*

Photo 3—*The Multiplex Engineering mux interface acts as a gateway between the vehicle network and anything with an RS-232 serial port.*

DANGEROUS CURVES AHEAD

That's not to say I've completely mastered all the mysteries. If you look closely at the program, you'll see I had to resort to a bit of ad hoc hacking to make it work. In particular, I discovered that it was best to adopt a policy of listening only for the stuff I wanted to hear.

For example, when sending a command from the SBC, I discovered the Multiplex Engineering unit isn't happy if you dawdle between bytes and will complain by sending back an error message. I was able to stop that by streamlining the inner loop of code that sends the message (by pre-computing the checksum). Considering that the BASIC SBC I'm using is fairly fast (it's a compiled, rather than interpreted, BASIC), watch out for this if you're using something slower.

The good news is I was no longer getting back stuff when I didn't expect it. The bad news is I was getting some stuff back that I didn't expect. Every now and then the RPM came back 0000. Curiosity aroused, I probed further to take a closer look at the incoming messages.

Sure enough, as shown in Figure 2, some strange activities were taking place. In particular, every now and then I'd get an odd packet that apparently is a valid diagnostic response as far as J1850 is concerned, but certainly doesn't contain the proper RPM info. While you're thinking about that, also note that even in the valid RPM packets, there's a third mystery byte following the two RPM bytes called for by the SAE spec. Following the if-in-doubt, throw-it-out approach, I finally got my digital J1850 tach working like a charm.

What does all this mean? Why do car dealers buy so many balloons? Seems to me the car biz is just bizarre, both at the showroom and under the hood.

WWW.MYWHEELS.CAR?

I though historically resistant to change, I think the silicon revolution will inexorably work it's magic on the car business. The manufacturers will realize that the car network isn't about the old-fashioned proprietary, dealer-only strategy. Rather, as with PCs, it is a great platform for neat features and services that they (and third-parties) will be glad to provide and encourage eventually.

What might the future hold? Remote diagnosis and even repair. Car black boxes that eliminate courtroom finger-pointing after an accident. A key for teen drivers that doesn't let them burn rubber. Dial-up smog checks in which the car testifies to it's own cleanliness (no need to pay for a tailpipe proctologist). Stolen cars that not only report their location, but stop running or, for kicks, automatically chauffeur the miscreant to the police station.

You may laugh, but click over to the Dearborn Group to check out their Gryphon Ethernet & TCP/IP Road Wide Web server.

I realize all this sounds rather visionary (as in "Tom's having visions again…"), but when it comes to silicon it's never a question of if, only when—and whether it comes in leather. ■

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for several years. You may reach him by e-mail at tom.cantrell@circuitcellar.com or by telephone at (510) 657-0264.

RESOURCES

Diagnostic RS-232 Multiplex Interface
Multiplex Engineering
(805) 964-6802
Fax: (805) 964-0890
www.multiplex-engineering.com

OBD2Scan PC-based Scan Tool
Baum Tools Unlimited, Inc.
(800) 848-6657
Fax: (941) 927-1612
www.baumtools.com

AutoTap PC-based Scan Tool
B&B Electronics
(815) 433-5100
Fax: (815) 433-5105
www.autotap.com

Ease PC Scan Tool
Ease Simulation, Inc.
(570) 465-9060
Fax: (570) 465-9061
www.easesim.com

Vehicle Control Module (VCM)
Delphi Automotive Systems
(248) 813-2000
Fax: (248) 813-2670
www.delphiauto.com

Gryphon Ethernet & TCP/IP Multiplex Server
Dearborn Group Inc.
(248) 488-2080
Fax: (248) 488-2082
www.dgtech.com

CIRCUIT CELLAR Test Your EQ

Problem 1—If you apply a sinewave stimulus to a linear time-invariant circuit, its response is also a sinewave. A squarewave can be shown to be the sum of a set of sinewaves, so by superposition, the response of the same circuit to a squarewave stimulus should also be a squarewave. What's wrong with this logic?

Problem 2—The energy stored in a capacitor is defined to be $E = (V^2/2)\,C$.

Suppose you have two 1 F capacitors, one is charged to 1 V, and the other is discharged. The energy in the charged capacitor is:

$$(1V^2/2) \times 1\text{ F} = 1/2\text{ J}.$$

Now, you connect the second capacitor across the first, effectively creating a capacitor of 2 F. The charge redistributes itself across the doubled capacitance so that the voltage on each capacitor is now 1/2 V. However, the energy in the system now evaluates to $((1/2\text{ V})^2/2) \times 2\text{ F} = 1/4\text{ J}$. What happened to the missing energy?

Problem 3—An embedded 16-bit microcontroller is connected to an 8-bit DAC, where the DAC is connected to the high-order half of the data bus. What is the purpose of *y* in the following driver?

```
#define DAC (*(unsigned *) 0x8000)

void DAC_out (unsigned x)
{
    static unsigned y;

    x += y;
    y = x & 0x00FF;
    DAC = x;
}
```

Problem 4—What is the Miller effect, and why is it generally considered a bad thing? How is it sometimes used to advantage?

What's *your* EQ?—The answers and 4 additional questions and answers are posted at www.circuitcellar.com.

You may contact the quizmasters at eq@circuitcellar.com.

8 more EQ questions each month in Circuit Cellar Online see pg. 2

PRIORITY INTERRUPT

Choice Versus Default

I suppose that everything we do can be rationalized to being a matter of choice or default. Do we bother to learn a new task or expand our expertise if we don't have to? How significant must the benefits of doing something a new way be for us to make an effort to learn it?

Let me share a humorous example. My 10-year-old grandson bounded into our house recently. After the usual hugs and pleasantries, he reverted to being a typical adolescent. After conning me out of new batteries for his Gameboy, he piped an urgent imperative, "What time is it? I can't miss Stone Cold Steve Austin!"

I looked at the wall clock that was barely 15′ away and noted the time was 4:37 p.m. Becoming a facilitator for the WWF wasn't something I would do easily. I pointed at the wall clock rather than simply answering. He looked at the clock. Surprisingly, instead of reading the time, he jumped up and ran into the guest bedroom where he usually stays when visiting. At the conclusion of his round-trip dash, he plopped back in the kitchen chair and said with a sigh of relief, "It doesn't start for another hour. I hope we're having dinner early."

Having kids around is a new thing for me. People who survived parenthood warned me that I should neither react immediately to what kids say nor look at every situation like it was supposed to be a learning situation for the kid. As for the WWF and me, we'll have to remain in a state of mutual coexistence.

It only took me a few seconds to realize what was going on here. The wall clock I pointed at in the kitchen is analog—the big hand and little hand deal. The clock in the bedroom is digital. Was the answer that simple? Trying not to be the condescending grandparent, I pointed at the kitchen wall clock and said, "Hey, kid?" (I affectionately called him kid when I'm trying to make a point), "Can you read that?"

"If I study it, I can usually figure it out. I just thought I'd save time by reading the clock in the bedroom. I can't miss Steve Austin, you know," he answered.

I won't bore you with the details after that. Let's just say that it demonstrated his level of necessity wasn't great enough to choose an unfamiliar technique when, with a little more effort, there was a default approach. Unfortunately, this lesson made me consider whether or not my concept of choice versus default needed modification too.

At least for electronics people, this experience parallels using digital versus analog design solutions. I don't mean obvious digital functions like encryption or combination logic. I mean real analog I/O, traditionally done with external analog circuitry attached to a processor, versus synthesized analog functions using mostly digital means. A typical example of this is a ramp generator. You can do it with an op-amp. You can also do it with a processor and a DAC. The question is, which method would you choose these days, and what issues influence your decision?

Just like calculators have reduced tedious hand calculation, today's technology strives to meet the demand. Newly designed programmable analog architectures attempt to make analog interfacing more comfortable for digital designers. The only saving grace is that at least for now, programmable components aren't presented as a way for designers to avoid understanding real analog interfacing issues. Not everything can nor should be done with digitally programmable devices. Things like signal amplification and filtering are still less expensive and less power hungry using an analog approach.

Design architecture is not black or white. Cost and design finesse may not even enter the equation. These days, often the only necessity is product delivery. Over the years, I've been critical of the fact that fewer engineers have traditional analog design experience. I also believe there is a tremendous gray area in design issues that can justify virtually any approach that succeeds (that's the only way I can rationalize using an embedded PC where an 8051 could do the job).

Nonetheless, design technique should be an informed choice and never a consequential default. Our subtitle is "The Magazine for Computer Applications". Sometimes this means describing analog, as well as digital designs. Other times it means describing overkill instead of finesse. Regardless of the subject, I want you to know that our real message is always helping you have an educated choice rather than merely the default.

steve.ciarcia@circuitcellar.com