www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#110 SEPTEMBER 1999

EMBEDDED APPLICATIONS

Implementing Voice Recognition

Sprucing Up HCS-II

PIC18C*xxx*—
A New Micro Poised for Action

Inside SmartMedia,
the Latest in Memory Cards

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.

WWW.CIRCUITCELLAR.COM/ONLINE
Table of Contents for August 1999

CONNECT YOUR PIC TO THE INTERNET
INTERNET PIC® 2000 CONTEST
NOW, GETTING CONNECTED TO THE INTERNET CAN EARN YOU CASH
www.circuitcellar.com/pic2000

INSIDE ISSUE 110

EMBEDDED PC

TASK MANAGER

On the Home (Automation) Front

It's easily one of the most popular *Circuit Cellar* newsgroups. Focused on the HCS-II, the home automation system developed by Steve, Ken Davidson & Co. in the early '90s, the cci.hcs2 newsgroup is a hotbed of advice and information on this popular system.

I wonder what the draw is. Is it the ability to get advice from people in the know? Is it the atmosphere of online camaraderie? Or is it just that people like to relate their own experiences?

For years, the HCS and HCS-II have served as a source of *Circuit Cellar* articles because people have wanted to share their stories. For example, one author, John Morley, has told us about his solid-state barometer, a wind direction and speed monitor, as well as an automatic lawn sprinkler controller (check your bookshelves [or back-issues CD-ROMs] for *Circuit Cellar* 63, 68 & 80).

It's been more than features, though. In fact, HCS introduced our first MicroSeries column in 1996 ("Applying the HCS II," *Circuit Cellar* 77 & 78), when two of its creators described how they'd taken the system further. Ken started us off in Part 1 with a look at simulated activity (e.g., "smart" automatic lighting), and the following month, Steve got us interested in how he built a weather station using Micromint's Answer MAN network module.

More specifically, Steve got Mike Baptiste interested. According to Mike, he has always enjoyed learning about electronics and home automation, but Steve's article tipped the scales. Now he just had to do something about it.

Although he has been a significant contributor to the discussions on the cci.hcs2 newsgroup for a long time now, and although we've been able to hear about how things have gone in Mike's own HCS-II installation, we haven't been privy to all the in-depth stories. There just isn't the opportunity or inclination to type everything out in a newsgroup situation. As you may know, it's more of a back-and-forth Q&A experience.

In other words, what hasn't been seen on the cci.hcs2 newsgroup yet is the kind of in-depth information presented in Mike's new Embedded Living column. Every other month, he will be bringing us the whats, the whys, and more importantly, the hows of implementing home automation. To begin the column, this month, Mike relates how he made his own network modules.

Mike's column reaffirms *Circuit Cellar*'s commitment to home-automation coverage. As you probably well know, *Circuit Cellar* is by engineers, for engineers—in other words, by the readers, for the readers. If you decide to implement some new part of your home automation system and want to tell us how you did it, do get in touch (editor@circuitcellar.com). The welcome mat is always out.

Eli

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER Steve Ciarcia	**ASSOCIATE PUBLISHER** Sue Skolnick
MANAGING EDITOR Elizabeth Laurençot	**CIRCULATION MANAGER** Rose Mansella
TECHNICAL EDITORS Michael Palumbo, Rob Walker	**CHIEF FINANCIAL OFFICER** Jeannette Ciarcia
WEST COAST EDITOR Tom Cantrell	**CUSTOMER SERVICE** Elaine Johnston
CONTRIBUTING EDITORS Mike Baptiste, George Martin, Ingo Cyliax, Bob Perrin, Fred Eady	**ART DIRECTOR** KC Zienka
NEW PRODUCTS EDITOR Harv Weiner	**GRAPHIC DESIGNER** Jessica Nutt
PROJECT EDITOR Janice Hughes	**ENGINEERING STAFF** Jeff Bachiochi, Ken Davidson, John Gorsky

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.
ARTICLE FILES: ftp.circuitcellar.com

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

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

NEW PRODUCT NEWS

Edited by Harv Weiner

WIRELESS RADIO MODEMS

The **MT and MS Radio Modems** are designed for data transmission in a local environment (e.g., a factory). Their range is typically ¾ mi. (line of sight) with a standard stubby antenna. Real-time, commercial, and industrial applications include data collection, industrial scales, crane control, computer cable replacement, and manufacturing systems (PLCs).

The modems operate in point-to-point, network, or multidrop (RS-422/-485) mode. Point-to-point mode replicates the use of an RS-232 “extension cord,” and network mode allows the connection of up to 99 remote radios with a single network controller. Multidrop is a drop-in replacement for the dual twisted pair used with many manufacturers’ multidrop protocols.

MT and MS radios operate in the UHF narrow band (450–470 MHz). FCC site licensing enforces interference-free use. Transmit/receive frequency stability is better than ±5 ppm. Transmitter output power is typically 120 mW. Receiver sensitivity is 0.5-µVdB/12-dB SINAD or better. Operating temperature range is from –30°C to +60°C.

Modems are available in portable and fixed mounted radios. The portable version’s built-in batteries provide over 8 h operating time. Higher gain antennas are available to extend range or overcome unusual obstructions.

Pricing starts at **$895** for a point-to-point radio.

Monicor Electronic Corp.
(954) 979-1907 • Fax: (954) 979-2611
www.monicor.com

SPEECH-RECOGNITION CHIP

The **RSC-364** is a complete speech-recognition system-on-a-chip for consumer electronic products and telephony applications. The system uses Sensory Speech 5.0 technology. With the RSC-364, a complete high-accuracy speech-recognition system can be built for less than $5.

Sensory Speech 5.0 technology, as implemented in the RSC-364, includes new noise-immune algorithms, new speech technologies, and significantly improved recognition accuracy. Current algorithms deliver accuracy rates of greater than 97% for speaker-independent recognition and greater than 99% for speaker-dependent recognition in various noise environments.

Sensory Speech 5.0 features Fast Digit Recognition, providing speaker-adaptive recognition for consecutive-digit dialing applications, where accuracy requirements per digit must be above 99.5%. Sensory’s 5.0 algorithms reduce response time to under 290 ms with a 32-word speaker-dependent vocabulary.

The RSC-364 has a 4-MIPS microcontroller, audio preamplifier, ADCs and DACs, watchdog timer, 64 KB of ROM, and 2.5 KB of RAM. The 8-bit microcontroller provides programming flexibility while supporting a complete suite of speech and audio technologies including speech recognition, speech and music synthesis, speaker verification, and audio record and playback. The RSC-364 permits on-chip storage of speaker-dependent or -independent vocabularies.

The device sells for under **$5** in quantities of 100,000.

Sensory, Inc.
(408) 744-9000
Fax: (408) 744-1299
www.sensoryinc.com

NEW PRODUCT NEWS

GPS ANTENNA

The **Skymaster** permanent-mount GPS antenna integrates a high-performance (26-dB gain) GPS patch antenna with a state-of-the-art, low-noise amplifier. It is packaged into a low-profile, extremely compact, fully sealed enclosure that is available in either black or white.

With its universal FME connector, Skymaster can easily attach to one of seven popular connectors: SMA, SMB, TNC, BNC, MCX, right-angle TNC, or right-angle BNC. A bulkhead mount with threaded nut, as well as an ample 5-m detachable cable, allows for quick and easy mounting.

Skymaster is 100% waterproof and designed to perform in hostile environments. This antenna features an operating temperature range of –30°C to +85°C in addition to strong bulkhead mounting.

Skymaster is priced at **$99**.

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

NEW PRODUCT NEWS

DIGITAL SOUND BOARD

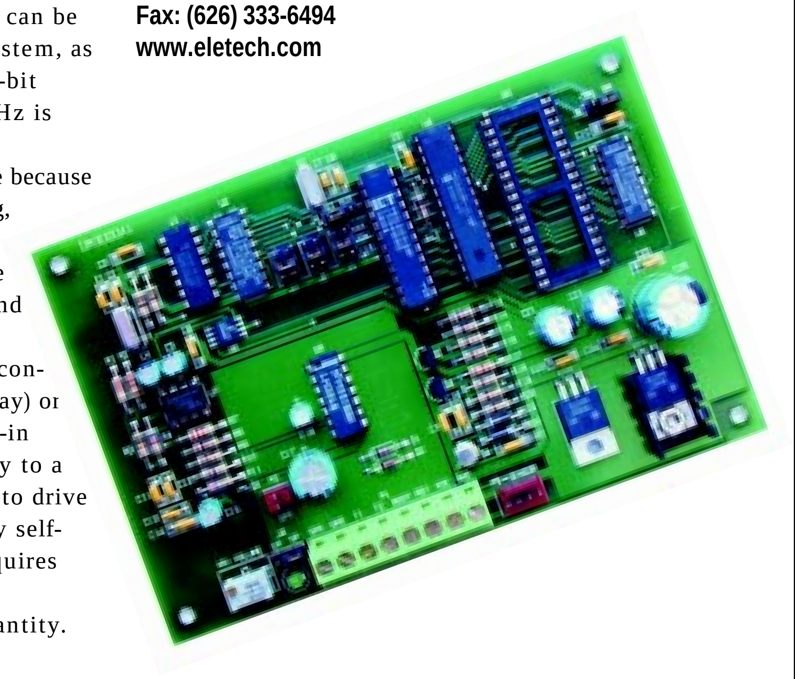
The **DM3000A Digital Sound Board** is Windows WAV compatible and can play back up to 32 sound segments stored in its onboard EPROM chip. Sound files can be created by using any digital audio authoring system, as long as they are in the WAV format. As well, 8-bit mono PCM sampling at 11, 22.05, and 44.1 kHz is supported.

The board provides the cleanest sound possible because the sound file is used directly for programming, without going through any extra digitization steps. Applications for the sound board include message repeaters, talking displays/exhibits, and amusement game sound effects.

Sound playback is activated by either a dry contact closure (e.g., a push-button switch or a relay) or a logic pulse from a microcontroller. The built-in power amplifier can deliver 5-W output directly to a speaker, or a line-level output can be obtained to drive an external power amplifier. The unit is totally self-contained on a 3.7″ × 5.6″ circuit board and requires only a single 12–24-VDC power supply.

The DM3000A is priced at **$78** in single quantity.

Eletech Electronics, Inc.
(626) 333-6394
Fax: (626) 333-6494
www.eletech.com



READER I/O

BULLS-EYE

Now you've gone and done it! The July issue takes the cake! I love robotics and you have ensured that I will keep subscribing to *Circuit Cellar*. The magazine is interesting and provides a variety of real-world projects and covers areas that other magazines rarely touch. I subscribe to all the major electronics magazines and all of them serve a useful purpose, but *Circuit Cellar* piques my interest every month.

Promise me (and the rest of us) more articles on robotics and the circuits used to interface them with the real world. "Electronic Odor Perception" by Silvio Tresoldi (*Circuit Cellar* 108) is outstanding and really helped me in my efforts to learn, develop, and have fun with mobile robots.

Would it be too much to ask for some articles dealing with software that controls robots? For example, some information on using encoders and how to actually make them work or how to use compilers and C/C++ to create hex files to be downloaded to microcontrollers.

If I continue to see issues like this, you will have a reader for life! Thank you for not forgetting the robotics crowd. Keep up the fine articles.

David Jackson
Pomona, California

FORGOT ONE

"Embedded OSs for Internet Appliances" by David Brooks in *Circuit Cellar* 107 was an extremely informative analysis of the Internet-appliance market. However, the discussion of available operating systems for embedded devices excluded one.

A company called e.Digital has developed a flash-memory file management system called the MicroOS. This file management system implements a small footprint in a digital device and it provides scalable functionality for editing and file management. The MicroOS is particularly suitable for applications involving dictation, voice, image, video and CD quality music.

MicroOS manages voice/video/data by means of compression. The maximum effectiveness of the OS is most noticeable when functioning with flash memory. Its architecture uses optimizable C code to manipulate the compressed data in flash memory.

The MicroOS is an architecture-independent OS, which means it can recognize and play a range of secure audio formats while reducing product development time. The MicroOS features a reduced chip count resulting in lower cost and power requirements for digital-device OEMs. For more information, visit www.edig.com.

James LaBoda
jimee11@hotmail.com

SETTING THINGS STRAIGHT

*Editor's note: A special thanks to Richard Johnson and Dale Yarker who pointed out some errors in Figure 2 of the "Low-Cost Software Bell-202 Modem" article (*Circuit Cellar *107).*

***[1]** The TX signal is shorted to the RING signal. **[2]** In the "RS-232 Transceiver" section, Q2 has its emitter and collector reversed. The emitter should connect to +5V. **[3]** Pin 2 of P1 connects to Q2's collector (after Q2 is corrected) and not at the diode as shown.*

Richard also suggested hooking up a large cap (22 µF) from the junction of R29 and D5 to ground to help the circuit's performance.

Dale commented that if the device won't go off-hook, the polarity of the line probably reversed somewhere between the telco and J3. To correct the problem, simply reverse the green and red wires. Other ways of making the hook switch insensitive to polarity would increase the parts count too much.

The updated code, an app note that includes a corrected schematic, and some information about licensing the DTMF code is now posted on the Circuit Cellar *web site.*

*Editor's note: Although there was no mention in the "Turn the Page" article (*Circuit Cellar *108) that code would be made available, the software for Ingo's pager project is now available for download via the* Circuit Cellar *web site.*

*Editor's note: In "Astronomical Issues" (*Circuit Cellar *108), Ingo mentioned a "freely available PCB layout program," but the URL was left out of the Sources section at the end of the article. Go to ftp.linuxppc.org/linuxppc/users/harry/PCB/ to download the software.*

Talking Back

FEATURE ARTICLE

Rodger Richey

Adding Speech to Embedded Applications

Training embedded apps to process speech may be as easy as finding the right 8-bit micro. Don't let what Rodger has to say about using an ADPCM algorithm and PWM output to generate speech go in one ear and out the other.

The ultimate form of feedback from a product is through speech. A product that reacts to stimuli with a verbal response is more likely to grab your attention than one without the capability.

In most cases, adding speech recording and playback requires extra processor bandwidth or an additional device such as a DSP or specialized audio processor. The cost, complexity, or lack of additional bandwidth, however, can prevent the speech features from being integrated into the product.

Now, if the words "8-bit microcontroller" were mentioned with respect to speech, some might chuckle to themselves, others might break into a fit of uncontrollable laughter, but certainly all would read on. Yes, it's true: a simplified Adaptive Differential Pulse Code Modulation (ADPCM) algorithm can be implemented on an 8-bit micro.

In this article, I explain the tradeoffs between bit rate and quality that are important in determining if you can use an 8-bit controller in the product. I also present the details of the origin as well as features of the ADPCM algorithm. Finally, I cover methods of integrating the microcontroller into the application as a speech encoder/decoder peripheral or as a complete speech-processing subsystem.

BIT RATE VS. QUALITY

When choosing a speech processor, you must first determine the desired quality of the speech reproduction. A speech-processing system attempts to balance the quality of the reconstructed speech with the bit rate of the encoding/decoding. In most cases, speech quality degrades as the bit rate drops.

The search for a happy medium between bit rate and quality has filled volumes. A high bit rate, high-quality speech processor implies a sophisticated algorithm that is computationally intensive with long encoding/decoding delays (i.e., requires the use of a DSP or special audio processor device).

This would also imply that an 8-bit microcontroller is not a solution for all applications but can provide reasonably good quality at medium-to-low bit rates. These tradeoffs between bit rate, quality, and system complexity can be summarized by the following questions:

- What level of speech degradation can be tolerated?
- What is the highest bit rate a system can tolerate (in terms of bandwidth)?
- What are the limitations on operating frequency, printed circuit board area, and power consumption?
- How much can you afford to spend on the speech subsystem?

Unfortunately, one answer can't satisfy all these questions. However, cost seems to drive most decisions.

Cost is the main factor behind bit rate. Lower bit rates are desirable because they lower operating bandwidth as well as memory storage requirements. It also means less mem-

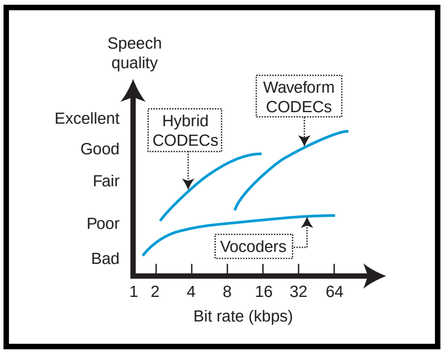


Figure 1—*A designer must make tradeoffs between bit rate and quality of reconstructed speech. After defining these two parameters, the selection of a speech coding algorithm can be made.*

ory to store, a fixed amount of speech, and lower cost. Figure 1 shows a graph of speech quality versus bit rate.

A typical system might sample speech with a 12-bit ADC at a rate of 8 kHz, which is more than sufficient to preserve signal quality. At this rate (i.e., 96 kbps), 1 min. of storage requires 720 KB.

To transmit the information over a communications channel requires something higher than 96 kbps to permit supplemental information (e.g., start-of-frame indicators, channel number). These requirements are beyond the scope of most applications and can be reduced by using speech coding.

Speech-coding techniques for reducing the bit rate fall into two categories. The first method is waveform coding.

There is a higher probability of a speech signal taking a small value rather than a large value. So, a speech processor can reduce the bit rate by quantizing the smaller samples with finer step sizes and the large samples with coarse step sizes.

The bit rate can be reduced further by using an inherent characteristic of speech—there is a high correlation between consecutive speech samples. Rather than encode the speech signal itself, the difference between consecutive samples can be encoded. This relatively simple method is repeated on each sample with little overhead from one sample to the next. An example of a waveform algorithm is ADPCM.

The other way to reduce bit rate is to analyze the speech signal according to a model of the vocal tract. The speech remains relatively constant over short intervals, and a set of parameters (e.g., pitch and amplitude) can define that interval of speech. These parameters are then stored or transferred over the communication channel.

This technique requires significant processing on the incoming signal as well as memory to store and analyze the speech interval. Examples of this type of processor (called a vocoder or hybrid coder) are linear predictive coding (LPC) or code-excited linear predictive coding (CELP).

Coder name	Algorithm type	Bit rate	MOS
G.711	log PCM	64	4.3
G.721	ADPCM	32	4.1
G.723	CELP	5.6 & 6.4	3.9
G.726	ADPCM	16, 24, 32, 40	–, 3.7, 3.9, 3.9
G.727	ADPCM	16, 24, 32, 40	–, 3.7, 3.9, 3.9
G.728	Low delay CELP	16	4.0
FS 1015	LPC-10	2.4	2.3
FS 1016	CELP/MELP	4.8/3.2	2.4/3.5
GSM	RPE-LTP	13	3.5
—	MBE	4.8	3.7

Table 1—*To help reduce the decision-making process, designers should rely on speech coder test results such as MOS, DAM, or SNR. Typically, the lower bit rate algorithms are significantly more complex than the higher bit rate ones.*

Quality is difficult to define or even measure. The goal of a measurement is to completely describe the quality of a speech processor in a single number. This measurement should be reliable across all measurement platforms as well as speech algorithms.

Unfortunately, however, measurements are broken up into subjective and objective. Subjective tests measure how a listener perceives the speech. Objective tests compare the original speech against the reconstructed output and make measurements based on signal-to-noise ratio (SNR).

The goal of a subjective test is to represent a listener's personal opinions about the reconstructed speech in a single number. The listener evaluates speech segments based on the intelligibility or signal degradations (e.g., nasal, muffled, hissing, buzzing). Several subjective tests exist such as diagnostic rhyme test (DRT), mean opinion score (MOS), and diagnostic acceptability measure. Table 1 shows the MOS score and bit rate for some common speech processors.

As I said, objective testing usually involves SNR measurements.

SNR is a measurement of how closely the reconstructed speech follows the original signal. The signal is broken up into smaller segments, and the SNR is measured. All the SNR measurements are averaged together to get an overall SNR measurement for the speech signal.

Although this measurement is sensitive to variations in gain and delay, it can't account for the properties of the human ear. The input to the speech processor is usually a sine wave or narrow-band noise waveform to maintain a repeatable test for all systems.

Because determining the quality of the speech processor is not as easy as picking the best number, both kinds of tests should be used to identify the best processor for your application. The best method may be to sit and listen to the outputs of the speech processor and simply select the one you like the best. After all, quality is not a measured parameter but rather a listener-perceived parameter.

WHAT IS ADPCM?

ADPCM is a waveform coding technique that attempts to code signals without any knowledge about how the signal was created. This implies that a waveform coder can be applied to other forms of data besides speech (e.g., video). In general, these coders are simple, with bit rates above

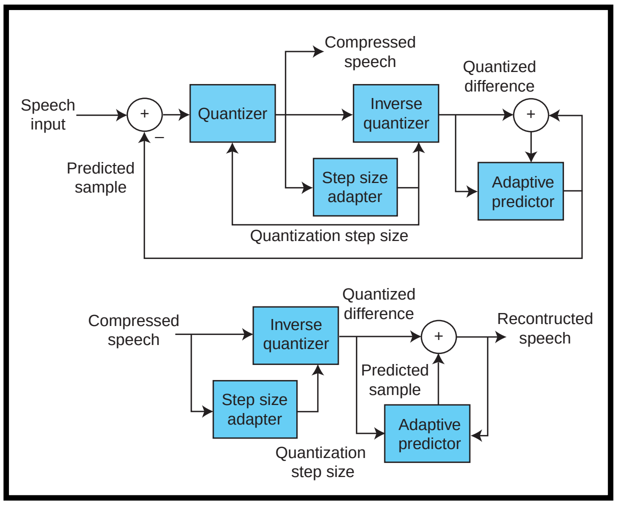


Figure 2—*Because the decoder block is embedded in the encoder, the ADPCM algorithm does not need to send or store any additional side information with the compressed data.*

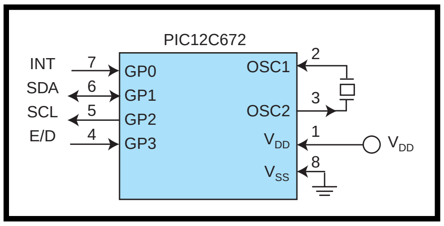


Figure 3—*The PIC12C672 provides the smallest solution for a serial coder peripheral. In addition to the I²C signals SDA and SCL, this device features an interrupt and encode/decode select signals.*

16 kbps. Anything lower degrades the reconstructed speech.

ADPCM is based on two principles of speech. Because there is a high correlation between consecutive speech samples, a relatively simple algorithm could be used to predict what the next sample might be, based on previous samples.

When the predicted sample was compared to the real sample, it was found that the resulting error signal had a lower variance than the original speech samples and could therefore be quantized with fewer bits. It was also found that no side information about the predictor would have to be sent if the prediction was based on the quantized samples rather than on the incoming speech signal.

The result was differential pulse-code modulation, formerly named ITU-T G.721. Further studies showed that if the predictor and quantizer were made to be adaptive (i.e., that smaller samples are quantized using smaller steps and larger samples with larger steps), the reconstructed speech more closely matched the original speech.

This adaptation helps the speech processor handle changes in the incoming speech signal more effectively. Thus the creation of ADPCM standardized to be ITU-T G.726 and G.727. Figure 2 shows a diagram of the encoder and decoder portions of ADPCM. Note that both the encoder and decoder share the same quantizer and predictor.

Most DSP manufacturers can show some type of speech algorithm that has been implemented for their architecture. Very few 8-bit microcontroller manufacturers can say the same, due to the horsepower required to implement the speech coding algorithms.

The ADPCM algorithm discussed in this article was developed by the now defunct Interactive Multimedia Association (IMA) based on an Intel DVI variation of the standard G.726. Normally, this algorithm is quite rigorous in the computation category, but the IMA version reduces the floating-point math and complex mathematical functions to simple arithmetic and table lookups.

A 16-bit 2's complement speech sample is converted into a 4-bit ADPCM code. The algorithm uses about 600 words of program memory and 13 bytes of data memory. Almost any 8-bit microcontroller can implement this algorithm, thanks to the small amount of resources required.

The code available for this article gives the complete ADPCM encode and decode routines written for use in Microchip's assembler (MPASM). The missing piece to the source code is that before each message is recorded or played, all the registers (`PrevSampleL`, `PrevSampleH`, and `PrevIndex`) must be cleared.

PERIPHERAL SPEECH CODER

A simple encoder/decoder peripheral can be implemented around a PIC12C672 or a '16C556A. The first thing to consider is the communication interface between the PIC and the main processor.

Lower end micros don't have any type of serial or parallel peripherals but they can be easily implemented in firmware. The complete code shows routines that can perform I²C, SPI, and RS-232 communications with a host processor, and Figure 3 shows a

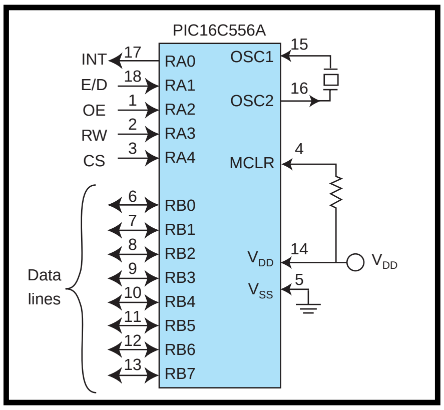


Figure 4—*The PIC16C556A provides a cost-effective parallel-interface solution to a speech coder peripheral. In addition to the standard parallel interface signals, it provides an interrupt and encode/decode select signals.*

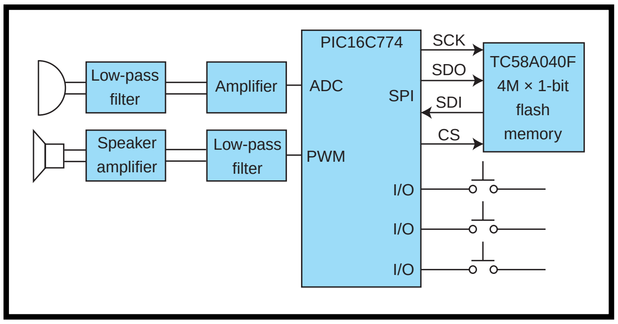


Figure 5—*For those applications requiring a complete speech-processing subsystem, the PIC16C774 with integrated 12-bit ADC, SPI, and 10-bit PWM provides the most integrated solution.*

block diagram for an I^{2}C implementation on a PIC12C672.

Because the microcontroller implements the serial interface in firmware, the application must ensure a good handshaking method to keep the micro from overflowing. A parallel interface routine is much easier to develop than the serial protocols, and Figure 4 shows an example of the parallel interface to a PIC16C556A.

The master I^{2}C routine uses approximately 77 words of program memory and 5 bytes of data memory. MPASM must be used to assemble this file.

One consideration when designing a system based around this routine is the transfer rate. If the PIC is the master of the interface, then the transfer rate is solely determined by the clock source to the microcontroller. If the PIC is a slave on the interface, then the transfer rate depends on the clock source as well as the firmware overhead to sample the incoming data.

The SPI slave routine uses approximately 16 words of program memory and 2 bytes of data memory. The same consideration concerning clock rate applies to this routine as well. Because of the overhead of sampling the SDI pin, the maximum clock frequency for the SPI slave is at least 18 instruction cycles, where one instruction cycle is the oscillator frequency divided by four.

The RS-232 routine uses approximately 54 words of program memory and 3 bytes of data memory. Although you should check to make sure that the micro has plenty of overhead, the transfer rate of RS-232 is usually much less than the PIC's oscillator frequency.

This routine only requires the user to define the oscillator frequency and the transfer rate. Several equations allow MPASM to calculate the necessary delays for bit times.

After the communication protocol is chosen, you have to put all the pieces together. First, you need to implement some type of data request from the main processor to the micro (for master) or from the PIC to the main processor (for slave).

The micro must control the flow of data to/from the main processor because the communication interface is implemented in firmware, not hardware. Otherwise, data may be lost. For a slave implementation, a single I/O line from the PIC connected to an external interrupt pin on the host processor easily accomplishes this.

The other important piece of information is the type of operation to be performed: encode or decode. This step can be accomplished two ways. First, a unique command from the host processor to the microcontroller can set the operation to follow. The host processor then initiates an encode or decode sequence by sending the command for encode or decode.

For an encode sequence, the host sends two 16-bit, 2's complement samples to the PIC. The PIC then responds with two 4-bit ADPCM codes packed into one byte. A decode sequence reverses the order. One byte of ADPCM codes is sent to the PIC, which responds with two 16-bit, 2's complement samples.

The second method is to use an I/O line from the host to the PIC to indicate an encode operation (I/O pin pulled low) or a decode operation (I/O pin pulled high). Note that encode and decode operations should not be mixed together.

All of the data to encode or decode should be sent consecutively to the micro. Once all of the data is processed, the host processor can change the type of operation to be performed.

This requirement is due to the fact that the ADPCM algorithm processes the next data based on previous data. Anytime the operation is switched, the encoder or decoder is initialized to a cleared state.

One other consideration is the selection of clock source to drive the PIC. The PIC's oscillator structure is flexible so either an external clock from the host processor or a local oscillator can be connected to it.

If your application has one system clock that drives all devices on the board, this same signal can be driven into the oscillator input on the PIC. Otherwise, a standard oscillator circuit can be used to provide the clock signal.

SPEECH SUBSYTEM

You can also use a PIC as a complete speech-processing subsystem. The PIC16C77*x* devices are ideal for this because of the 12-bit ADC and 10-bit PWM peripherals. The new PIC-18C*xxx* architecture can implement stereo record and playback at an 8-kHz sample rate because of its optimized instruction set, architecture, and 40-MHz operation.

The PIC can communicate to the host processor via any serial interface or even a simple keypad that implements play, record, next message, and previous message. Figure 5 shows a simplified block diagram of the speech subsystem based on a PIC16C77*x*.

The microphone input must be both filtered and amplified before entering the microcontroller. This input might be designed in two stages.

First, an amplifier stage with some limited automatic gain control provides between 40 and 60 dB of gain. The filter stage might be a fourth-order filter centered at 4 kHz for an 8-kHz sample rate. The PIC samples the incoming signal at 8 kHz and compresses the 12-bit sample down to four bits.

The memory size is determined by the amount of record time desired. At 8 kHz, the system requires 4 kbps of storage (8000 samples/s × 4 bits/sample). Therefore, 1 min. of record time requires 240 KB.

An ideal match for this type of system is the Toshiba TC58A040F 4M × 1 NAND flash-memory device. It stores approximately 131 s of speech at an 8-kHz sample rate and uses SPI as the communications interface.

You now have a choice to make on the speech output circuit. Although a DAC makes sense in some applications, the PIC's onboard 10-bit PWM peripheral can also be used to lower cost without giving up quality.

Admittedly, the DAC has better quality, but with the right filtering, the PWM module can provide good results. This filter can be a fourth-order filter centered at 4 kHz (and can be a copy of the input filter).

The final circuit—the speaker amplifier—is extremely application dependent. You may want to drive a speaker or a set of headphones. Many companies, like National Semiconductor and TI, make amplifiers specifically for driving speakers or headphones.

TALK ABOUT POTENTIAL

Although some applications need high bit rate and high-quality speech algorithms, most can use one like mine. Don't underestimate the power of an 8-bit micro. Given the right device, the medium bit-rate algorithms can be implemented successfully without a DSP or specialized audio device.

Improvements to the 8-bit architecture, operating speed, instruction set, and memory sizes have allowed the migration of low-end DSP applications to the 8-bit world. If you've never used a PWM module to generate speech, try it. You might be surprised.

Rodger Richey has worked for Microchip for more than four years in principal engineer and senior applications engineer positions. You may reach him at rodger.richey@microchip.com.

SOFTWARE

The software for this article may be downloaded via the *Circuit Cellar* web site.

REFERENCES

N.S. Jayant and P. Noll, *Digital Coding of Waveforms, Principles and Applications to Speech and Video*, Prentice Hall, Englewood Cliffs, NJ, 1984.

P.E. Papamichalis, *Practical Approaches to Speech Coding*, Prentice Hall, Englewood Cliffs, NJ, 1987.

IMA Compatibility Project, *Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems*, V.3.00, Oct. 1992.

R. Richey, *Adaptive Differential Pulse Code Modulation using PIC-16/17 Microcontrollers*, AN643, Embedded Control Handbook, Microchip Technology, 1996.

J.D. Tardelli, E.W. Kreamer, P.A. La Follette, and P.D. Gatewood, *A Systematic Investigation of the Mean Opinion Score (MOS) and the Diagnostic Acceptability Measure (DAM) for Use in the Selection of Digital Speech Compression Algorithms*, ARCON, www.arcon.com/dsl/sl24a.html.

RESOURCES

Comp.speech FAQ, www.speech.cs.cmu.edu/comp.speech

Dynastat Inc., *Subjective Testing and Evaluation of Voice Communications Systems*, www.bga.com/dynastat/index.html

T. Robinson, *Speech Analysis*, Cambridge University Engineering Dept., Speech Vision Robotics Group, www-svr.eng.cam.ac.uk/~ajr/SA95/SpeechAnalysis/SpeechAnalysis.html

J. Woodard, *Speech Coding*, Dept. of Electronics and Computer Science, University of Southampton, rice.ecs.soton.ac.uk/speech_codecs/index.html

SOURCES

PIC12C672, '16C556A, '16C774
Microchip Technology, Inc.
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

Amplifiers
National Semiconductor
(800) 272-9959
(408) 721-5000
Fax: (408) 739-9803
www.national.com

Texas Instruments, Inc.
(508) 236-3800
www.ti.com

FEATURE ARTICLE

Kock Kin Ko

Electric Vehicle Performance Analyzer

Ko takes us through the design steps of a Hitachi-based device that measures and analyzes electric-car speed, distance, fuel consumption, and travel time. All this information is available to the driver at the touch of a button.

Whenever I dropped in to see the staff in the electric car workshop at the Electrical Engineering Department of Ngee Ann Polytechnic, I left with an urge to get involved. I fancied the vehicle, and the principal lecturer was a good friend.

Although I didn't know much about the hardware stuff, our gurus knew every single electrical part of the vehicle and had even publicized the vehicle hardware a couple years ago in the *Straits Times* of Singapore. The hardware publicity was good, but no one ever wrote any embedded code to optimize the vehicle's performance.

Having driven a Ford Telstar for years, I looked at the bank of batteries packed on the electrical vehicle and had to ask myself if it was really economical to drive an electric vehicle. How could its performance be optimized?

Before figuring out how to optimize performance, I'd need a system analyzer to monitor or calculate its performance. This was my chance to get involved with the electric car project, and that's how I got started designing the performance analyzer.

The analyzer was developed using a Hitachi microcontroller. Three sets of performance data are collected in real time—fuel consumption (in amp-hours from the car battery) versus speed (in kilometers per hour), distance (in kilometers) versus speed, and time traveled (in hours and minutes, or minutes and seconds) versus speed.

Keys on the analyzer enable the driver to scroll through the recorded performance data. Any driver who starts a new trip can erase the performance data record or let the data accumulate over time and then download it to a PC at the end of the trip.

A graphical display of the performance data is shown on the PC screen. Indirectly, this display helps drivers convert their thinking from miles-per-gallon of gasoline to hours-per-amp of battery power. It also helps them estimate how many amp-hours are required to travel to a destination, or the optimum speed with the least amp-hours spent.

In this article, I present the software design of the analyzer and cover both the microcontroller and PC software (both in C). The micro's software was developed with a Hitachi C compiler, and the software for the PC was

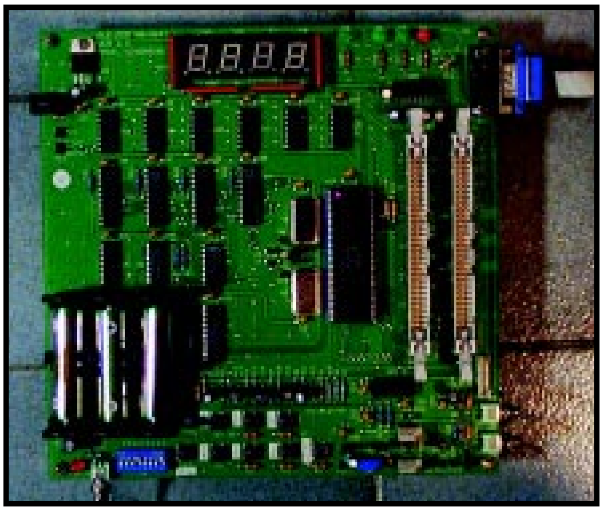


Photo 1—*Here's the prototype (courtesy of Hitachi Micro Systems Asia) that I used for software development. CPU H3644 is located at the center, the seven-segment display and LEDs at the top, button keys at the bottom, RS-232 cable at the upper right corner, and DIP switch in the lower left corner.*

developed with CVI (C for Virtual Instrument) from National Instrument.

The Hitachi H8/3644 8-bit micro used in this project is a dream machine for an embedded C programmer like me. It consists of 32-KB flash memory and a 1-KB RAM that's big enough for a C program. Sure, C is big, but I like its readability, portability, and reusability.

The H8/3644 has a number of special functions that I took advantage of. The Timer A interrupt provides real-time ticks in milliseconds. Timer B captures pulses generated from a rolling wheel, and analog channel 1 measures fuel consumption. Five external interrupts are used for sensing five keys, and the on-chip UART is used to download data to PC.

SYSTEM DESIGN

The four basic system measurements are real-time ticks in seconds (for measuring time), wheel pulses (for measuring distance), pulses per second (for measuring speed), and milliamps per second (for measuring fuel consumption). The Timer A interrupt schedules these four system measurements.

The real-time clock accumulates travel-time duration from seconds to hours. Distance is calculated by converting the total number of wheel pulses into kilometers. Speed calculation converts pulses per second into kilometers per hour. To calculate fuel consumption, milliamps per second are converted to amps per hour.

These conversions are done only on user request. When requested, the conversion results are displayed on a seven-segment display with four digits.

The raw speed data is accumulated to form statistical records. Speeds are divided into 13 ranges: <20 km/h, 20–30 km/h, 30–40 km/h, and so on up to >130 km/h. Timer A also schedules these accumulations.

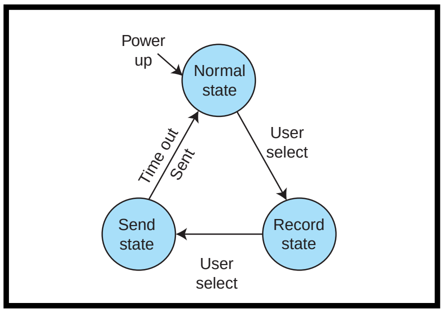
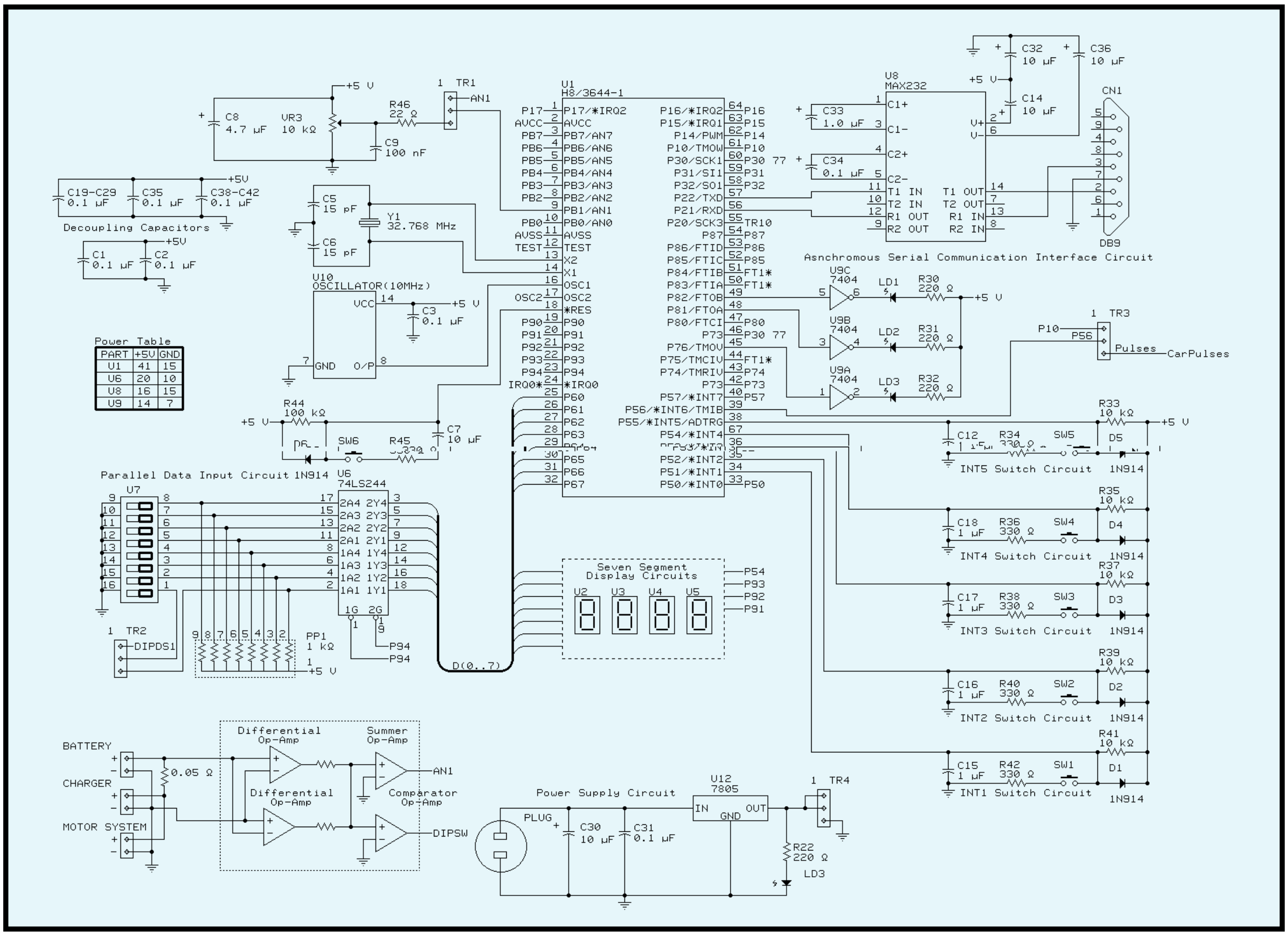


Figure 1—*The driver selects one of the three states available with the system. Data records are shown on the seven-segment display in the Record state and are sent to the PC in the Send state.*

The driver selects one of the three states available with the system—Record, Send, or Normal. Figure 1 shows the state transition diagram.

In the Record state, drivers can scroll through and view all records. In the Send state, the micro waits for the password to be sent from the PC before dumping the records to the PC. In the Normal state, drivers can view



Power Table

PART	+5V	GND
U1	41	15
U6	20	10
U8	16	15
U9	14	7

Figure 2—*Here's the schematic for the electric vehicle performance analyzer. Shown on the right-hand side of the CPU is the circuitry designed for the RS-232 connection, the LEDs, the car pulse input, and the button keys. The circuitry on the left-hand side includes the analog input, the seven-segment display, and the current measurement.*

S/N	File name	Purpose	CPU dependent
1	Adlpfc.c	Low-pass filter to average 32 samples	No
2	Car. c	Main program	No
3	DispMesg.c	Routines for showing all messages on four-digit display	No
4	Elecpara.c	Gets electrical system configuration and calculates conversion gain factor	No
5	Fourdigi.c	Algorithm for fitting any four-byte variable onto a four-digit display	No
6	Hardware.c	All hardware-specific codes including CPU initialization, driver for seven-segment display	Yes
7	INTR.c	All interrupt routines including Timer A interrupt, ADC interrupt, and INT external interrupts	Yes
8	Math.c	Converts short integer into five BCD digits	No
9	Serial.c	Routines for serial communication with PC	Yes
10	Statdata.c	Routines for defining and updating statistical records	No
11	UserIF.c	Routines for scrolling through menu for user selection	No

Table 1—*Listed here are the 11 software modules and their purposes. Modules that are CPU-dependent are also indicated.*

current speed, total distance and time traveled, total fuel consumption, and energy left in the battery. Regardless of what state the system is in, data measurement is done in real time.

HARDWARE DESIGN

Each of the five keys of the user interface are connected to external interrupt pins (see Figure 2). The five external interrupts for sensing keypress are INT1–INT5. They sense driver selection of data for fuel consumption, distance traveled, speed, time traveled, and system states, respectively.

A four-digit seven-segment display is connected to a data bus (port 6) and a control bus (port 9 pins 1–4). Four 8-bit data latches (74LS273) are used for latching data for the display's four digits. To latch data into the data latch, set the control line high, place the data on the data bus and reset the control line to low.

A differential op-amp measures the current drawn from the battery by the motor system. The differential gain is scaled to provide a 5-V output at maximum system current.

With its input across a 0.05-Ω resistor, the op-amp output AN1 is connected to analog channel 1. The op-amp circuitry also provides a digital output (DIPSW1) to signal whether current is charging into or discharging from the battery.

An eight-way DIP switch provides the system configuration. The switch is connected to the data and control buses as a parallel input port. Bits 2–3 of the DIP switch indicate four possible maximum system-current loads, and bits 4–5 indicate four possible battery sizes. CPU software reads these 4 bits at powerup to determine the conversion gain factor for calculating system current and a new battery's original energy.

Some hardware aids make software debugging easier. For example, a 5-V variable source (VR3) is used during software debugging rather than the op-amp voltage output. Through a jumper (TR1), VR3 is connected to analog channel 1.

The DIP switch also provides debugging features. Bit 0 simulates current direction (charging or discharging). To simulate the change in speed during software debugging, bits 6–8 simulate eight possible speeds while Timer A output is jumpered through TR3 to the Timer B input. CPU software reads these bits in every timer tick to determine the number of pulses transmitted at the Timer A output.

Three LEDs (LD1, LD2, and LD4) are used for burn-in testing. Their continuous flashing indicates that the main program, the Timer A interrupt routine, and the ADC interrupt routine are alive.

SOFTWARE DESIGN

To maintain modularity and portability, I wrote 11 files. Table 1 lists the purposes of each. On powerup, the

Listing 1a—*`INT()` sets `INTx_menu_change` to indicate that a specific key was pressed.* ***b—****The main program `main()` calls the routine `user_change_menu` to determine user selection `op_menu`.*

a)
```
#pragma interrupt INT
void INT(void)
{
  if(IRR3.BIT.B1)              /*if interrupt flag set?*/
  {
    IRR3.BIT.B1=0;             /*clear interrupt flag*/
    INT1_menu_change = 1;      /*set semaphore*/
    new_menu = INT_1;
  }
  if(IRR3.BIT.B5)              /*similarly for INT2, INT3,and INT4*/
   {
    IRR3.BIT.B5=0;
    INT5_menu_change=1;        /*set semaphore*/
    new_menu = INT_5;
  }
}
```

b)
```
main()
{
  system_engine_start_up_sequence();
  do {
    user_change_menu();      /*determine op_menu*/
    switch (op_menu)
     {
      case INT1_MENU_ADC_VOLTAGE:
        adc_main();
        break;
      case INT1_MENU_CURRENT:
        current_main();
        break;
        …
     }
  }while(1);
}
```

main program, car.c, calls routines in hardware.c to initialize the CPU, routines in elecpara.c to determine system hardware parameters, and routines in statdata.c to initialize system variables (e.g., statistical records).

After interrupts are enabled, routines in INTR.c are called to handle the Timer A, ADC, and external interrupts. INTR.c also includes routines for updating the raw data for distance and speed. The lowpass filter routine in Adlpfc.c is called to get the average raw data for fuel consumption.

Besides scheduling data updates, Timer A also schedules when to update statistical records (done by calling routines in statdata.c). After initialization, the main program enters a forever do-loop. First, it calls routines in UserIF.c to determine what menu the user selected. Then, the main program executes routines in car.c to serve the user.

Routines in UserIF.c scroll through all the menus. These routines then call routines in DispMesg.c to dump menu messages on seven segments.

To best serve the user, three things need to be done. Conversion routines in car.c convert raw data into distance, speed, and fuel consumption. The auto-scaling algorithm in Fourdigi.c scales large numbers (four-bytes long integer) into two-byte short integer. BCD conversion routines in math.c convert the short integer into five BCD digits.

Finally, the auto-scaling algorithm puts the five BCD digits on a four-digit display with the decimal point automatically shifting across the display to indicate the scale of the data. If the user sends the data to a PC, serial.c handles the serial communication.

USER MENU

Each key has a menu that consists of a sequence of selection choices. Pressing and holding a key enables the user to scroll through the selection choices.

The INT1 menu offers these options: A (analog voltage from the ADC), Curr (current drawn from the battery by the motor system), AH-d (total amount of amp-hour discharging from battery), AH-C (total amount of amp-hour charging to battery), and Engy (remaining energy stored in the battery).

The INT2 choices are Cout (number of pulses accumulated for 1 km), onE (1 km dial, accumulated distance up to 1 km), and diSt (total distance traveled). The INT3 menu offers CPS (number of pulses per second generated by the rolling wheel) and SPEd (speed in kilometers per hour).

INT4 provides five choices—tinE (time in HH.MM or MM.SS), StAr (to start or continue to accumulate raw data), StoP (temporarily stops accumulating raw data), CLEA (clears all raw data and records), and oFF (off display and stop accumulating raw data). The three system states for the INT5 menu are rECd (enters record state), send (for sending records to the PC), and roFF (to turn the record state off and return to the normal state).

Listing 2—*user_change_menu calls change_INTx_menu() to scroll through the choices of INTx menu. The latter calls display_INTx_menu to display the menu choices.*

```
void display_INT1_menu(void)
{
  switch (INT1_op_menu) {
    case INT1_MENU_ADC_VOLTAGE :
      show_A();                                /*display A*/
      break;
    case INT1_MENU_CURRENT :
      show_Curr();                             /*display curr*/
      break;
    case INT1_MENU_AMP_HR_DISCHARGE:
      switch (record_data_display_on){
    case RECORD_MODE_ON:                       /*Record State on*/
      if (INT1_down)                           /*if key is held down*/
      {
       INT3_speed_index = inc_INT3_speed_index(INT3_speed_index);
       amp_hr_discharge_record_on(INT3_speed_index);
        INT1_down = 0;
      }
      else show_AH_d();  /*key is pressed but not yet held down*/
      break;
    case RECORD_MODE_OFF:                      /*Normal State*/
      show_AH_d();                             /*display AH-d*/
      break;
    }
    break;
  case INT1_MENU_AMP_HR_CHARGING :
    show_AH_C();                               /*display AH-c*/
    break;
  case INT1_MENU_ENERGY:
    show_Engy();                               /*display Engy*/
    break;
  }
  display(dispdat);                            /*to display on 7 segments*/
}

void change_INT1_menu(void)
{
  do{
    switch (record_data_display_on ) {
      case RECORD_MODE_OFF:                    /*in Normal State*/
        /* if 1st time to switch back to this menu, do not increase
        /*  INT1_op_menu. So that when switching between keys, user
        /*  menu preference is kept  */
        if (new_menu ==old_menu)INT1_op_menu += 1;
        else old_menu = new_menu;              /*update menu choice*/
        if (INT1_op_menu > INT1_MAXMENU) INT1_op_menu = 0;
         break;
      case RECORD_MODE_ON:                     /*in Record State*/
        INT1_op_menu = INT1_MENU_AMP_HR_DISCHARGE;
         break;
    }
    INT1_menu_change = 0;
    display_INT1_menu();
    wait_and_see_display();
    INT1_down = test_INT1_button_down();
    }
  while (INT1_down);     /*while mode button being held down*/
  op_menu = INT1_op_menu;
}
```

Listing 3—*The Timer A real-time scheduler schedules all events. Updating statistical records is done at real-time "*`quad_sec" = HALF_SEC`*, whereas raw data is done at real-time "*`quad_sec" = SEC.`

```
#pragma interrupt TIMA
void TIMA(void)
{
  IRR1.BIT.B6=0;              /*clear Timer A Interrupt hardware flag */
  if (adc_on ==1)
  ADSR.BIT.B7=1;              /*start ADC every 31.25 ms */
  real_time_tick++;           /*accumulate 31.25 ms into ticks */
  switch (real_time_tick) {
    case 16 :
      quad_second = HALF_SEC;
      if ((start_flag ==1)&&(sending ==0)) {
      update_statitics ();  /*update records only if not sending to PC */
    }                     /*to avoid sending half-updated data to PC*/
    break;
    case 32 :
      quad_second = ONE_SEC;
      real_time_tick = 0;
      if (start_flag ==1) {    /*INT4 key StAr menu sets start flag*/
      adc_average = lpf(adc_buf_ch0);/*moving average lowpass filer*/
      milli_amp_average = (unsigned long int) adc_average * max_elect_load;
      milli_amp_sec += milli_amp_average;/*raw data for fuel consumption */
      milli_amp_hour = milli_amp_sec / 3600;
      test_if_over_current(&adc_average);
      update_energy();             /*energy remained in battery */
      elapse_time[2]++;             /*seconds*/
      update_time(elapse_time);     /*raw data for time traveled */
      update_1sec_TMB_counts();     /*raw data for speed and distance */
    }
    break;
  }
  wk_PDR[8] ^= SET1;          /*toggle LED1 to indicate TMA alive */
  PDR8.BYTE = wk_PDR[8];
}
```

In the Record state, the user holds key INT1 to scroll through records of fuel consumption for different speed ranges. Key INT2 gives records of distance versus speed, and key INT3 scrolls through different speed ranges. Pressing key INT4 shows the user records of time traveled versus speed.

DETECTING USER SELECTION

The external interrupt routine `INT()` sets semaphore `INTx_menu_change` to indicate that a specific key was pressed (see Listing 1a). The main program `main()` calls routine `user_change_menu()` to determine the user selection `op_menu` (see Listing 1b).

When a key is held, `user_change_menu()` calls `change_INTx_menu()` to scroll through the choices of `INTx` menu (which then calls `display_INTx_menu()` to show the menu choices) (see Listing 2).

`op_menu` is set to a numerical value when the user releases the key. According to `op_menu` value `xyz`, `main()` executes `xyz_main()` as shown in Listing 1b.

Notice three things in Listing 2. First, in routine `change_INT1_menu()`, `new_menu` and `old_menu` are used to keep the user preferences by not increasing the `INTx_op_menu` value when the user switches back to an old key. This way, when switching between keys, the driver can keep his preference choice for each key.

Second, `display_INT1_menu()` puts up a display depending on the state the system in. In the Normal state, `AH-d` (for fuel consumption) is shown on seven segments. If the system is in Record state, the user can scroll through the records of fuel consumption versus speed by holding the INT1 key down. `amp_hr_discharge_record_on (INT3_speed_index)`is called to show the fuel-consumption records.

Note that `INT3_speed_index` increases as the user holds down the INT1 key to look at the fuel-consumption records. Should the user switch to INT2, the record of distance traveled at the latest speed index set before the key was switched is displayed. That's how the four keys (INT1–INT4) coordinate to show recorded data with respect to the same speed range.

REAL-TIME SCHEDULER

The Timer A interrupt schedules all events. Two main events are scheduled to take place at different time slots. Updating statistical records is done at real time "quad_sec" = HALF_SEC and updating the raw data is done at "quad_sec" = ONE_SEC, as shown in Listing 3.

Scheduled in real time, for every 31.25 ms, one ADC sample is saved in ring buffer adc_buf_ch0[32] by the ADC() routine in Listing 4a. At real time "quad_sec" = ONE_SEC, a low-pass filter routine is called to get the average of 32 ADC samples. This is done by adc_average = lpf(adc_buf_ch0);. This value is added into raw data milli_amp_sec (see Listing 3).

Timer B counts the pulses generated from the rolling wheel. At real time "quad_sec" = ONE_SEC, update_1sec_TMB_counts()reads the counts in Timer B and calculates the counts in 1 s (inc_count). The routine also accumulates inc_count into sum_inc_count. When sum_inc_count exceeds COUNT_PER_1KM, total_distance increases by 1 km (see Listing 4b).

At "quad_sec" = ONE_SEC, update_time() updates the time traveled from seconds into minutes and from minutes into hours. At "quad_sec" = HALF_SEC, the function update_statistics() is called to update statistical records. Raw data versus speed is calculated into a data structure called struct statistic_

Listing 4a—*ADC() stores an analog sample in a ring buffer.* ***b***—*Updating wheel pulses per second is done by update_1sec_TMB_counts().* ***c***—*Record format is defined by the data structure statistic_form.* ***d***—*Statistics are updated with respect to speed index in update_statistic().*

a)

```
#pragma interrupt ADC
void ADC(void)
{
  IRR2.BIT.B6=0;                          /* clear Interrupt hardware flag*/
  adc_cur_result=ADRR.BYTE;               /*read ADC result*/
  bptr++ = adc_cur_result;                /*put inside ring buffer*/
  if (bptr == (unsigned char )tail) {     /*pointer wraps around*/
    bptr = (unsigned char *)head;
    first_round_adc = 1;
  }
  wk_PDR[8] ^= SET2;                  /*toggle LED2 to indicate ADC alive*/
  PDR8.BYTE = wk_PDR[8];
}
```

b)

```
void update_1sec_TMB_counts(void)
{
  new_count = TCB1.BYTE;                  /*read Timer B*/
  inc_count = new_count - old_count;
  /*get difference for counts per second*/
  old_count = new_count;
  sum_inc_count += inc_count;             /*add into 1-km count*/
  if (sum_inc_count > COUNT_PER_1KM){     /*check if more than 1 km*/
    sum_inc_count -= COUNT_PER_1KM;
    total_distance++;                     /*total distance in kilometers*/
  }
}
```

c)

```
struct statistic_form {
  char elapse_time[3];
  unsigned long int milli_amp_sec;
  unsigned int total_distance;
  unsigned int sum_inc_count;
};
```

d)

```
void update_statitics (void)
{
  speed_index = get_speed_index(inc_count,speed_index);
  update_stat_distance_travelled(speed_index);
  update_stat_energy_consumpt (speed_index);
  update_stat_time_travelled(speed_index);
}
```

Listing 5a—`total_KM_main()` calculates the total distance. **b—**Data auto ranging is done by `standard_display()` to fit a BCD number onto a four-digit display.

a)
```
void total_KM_main(void)
{
  if (quad_second == ONE_SEC)
  {
    switch (record_data_display_on) {
      case RECORD_MODE_OFF:
        total_KM_record_off (sum_inc_count,total_distance);
         break;
      case RECORD_MODE_ON:
        total_KM_record_on (INT3_speed_index);
         break;
    }
    quad_second = 0;
  }
}
```

b)
```
char standard_display(unsigned long int *current_data, char scale)
{
  unsigned int current_int;
  scale = auto_scale_find (current_data, scale);
  current_int = scale_down(current_data,scale);   /*scale down to integer*/
  binary_BCD(&current_int, five_digit_BCD);
  four_digit_BCD_display_auto(five_digit_BCD,scale);
  return (scale);
}
```

form `car_data[13]`. The structure tag is shown in Listing 4c.

The index of structure array `car_data[ ]` is the speed index. It refers to different ranges of speed. `get_speed_index()` calculates the speed range (see Listing 4d).

CONVERSION ROUTINES

`Main()` calls `amp_hr_discharge_main()` to convert `milli_amp_sec` into `amp_hour`, `one_KM_main()` to convert `sum_ inc_count` into distance within 1 km, `total_KM_main()` to calculate total distance, and `speed_main()` to convert counts per second into kilometers per hour.

Each `xyz_main()` does different things for different system states. For example, `total_KM_main()` calls `total_KM_record_off()` in the Normal state to display the total distance travelled. In the Record state, `total_KM_record_on()` displays distance traveled versus a speed range (see Listing 5a).

`xxx_main()` calls `standard_display(&current_data,scale)` to do auto ranging. This routine first calls `auto_scale_find(unsigned long int *data, char previous_scale)` to find out the scale order of current data. Then it calls `scale_down(unsigned long int *data_long, char auto_scale)` to scale down the number according to the scale, so that the long-integer data can be fitted into a short integer.

Next, `binary_BCD(&current_int, five_digit_BCD)` converts the short integer into five BCD digits. `four_digit_BCD_display_auto(five_digit_BCD,scale)` puts the them on a four-digit display with the decimal point automatically shifting across the display to indicate the scale of the data. This code also blanks out leading zeroes on the display (see Listing 5b).

Autoranging for time is done by `display_time()`. The difference between the two time displays (HH.MM or MM.SS) is that the dot flashes four times per second in the HH.MM format and not at all in the MM.SS format.

In the Send state, the system counts down 6 s in `rx_byte()` while waiting for the password to be sent from the PC. `rx_sync()` checks the password `*idn?`. If the password is not received, the routine displays `tOut` (timeout); otherwise `send_val()` sends the entire structure `car_data[ ]` to the PC and displays `Sent`.

While waiting for my hardware prototype, I used a target board for debugging. I varied the 5-V voltage source to simulate changes in current consumption. I also changed the DIP switch setting to simulate changes in speed. The program runs on a Hitachi target board ALE TDS H8/3644 V.2.0 with an onboard flash H8/3644.

READY TO GO

Future improvements are definitely possible. I still haven't added the software to calibrate distance versus pulse count generated from the rolling wheels. And, data sent to the PC is not appended with `crc`.

I've finished debugging the prototype software and have permission to mount the analyzer on one of the vehicles, so system integration tests will begin soon. Now when I visit the workshop, I leave with a feeling of satisfaction—now I'm a part of the team. ■

I wish to thank Hitachi Singapore for their contribution to the project. Special thanks to Danny Wong, senior technical supporting engineer. Limited free samples of the flash H8/3644 can be obtained from Danny. He can also help burn `car.mot` into your free sample. You may reach him at dwong@has.hitachi.com.sg.

Before joining Ngee Ann Polytechnic, Kock Kin Ko worked as a principal engineer in Singapore's defense industries as an embedded designer. He was also an official CVI instructor appointed by National Instruments (Singapore). He is a software technology manager at Philips Singapore. You may reach him at koxiaozq@cyberway.com.sg.

SOFTWARE

Executable code is available via the *Circuit Cellar* web site.

SOURCES

H8/3644
Hitachi America, Ltd.
(800) 448-2244
(415) 589-4207
Fax: (415) 583-4207
www.hitachi.com

CVI
National Instruments (Singapore)
+65 226-5886
Fax: +65 226-5887
www.natinst.com.

EMBEDDED LIVING

Mike Baptiste

Expanding the HCS-II

Making Network Modules

To improve his new house, Mike added a home automation system. Then he started improving his home automation system. Fasten your toolbelt as he nails down the details in this new column, Embedded Living.

I've tinkered with electronics since I was a kid, but somehow I ended up working with software for a living. However, my heart has always been in hardware, so I decided to use home automation to satisfy my need to build stuff.

I started receiving *Circuit Cellar* when I was still in high school and have followed the progress of the HCS-II Home Automation System with much interest. Not much opportunity to automate a dorm room, but after college, I moved down south and bought a townhouse. Plenty to automate there, but a new job can take up lots of free time. If I was going to get an HCS-II, I had to buy a nice big house!

My wife would chuckle to hear that she's partly responsible for my home automation "hobby." After the wedding, we bought a large house in the country that we planned to gut and renovate. A home automator's dream!

About the same time, Steve wrote his Answer MAN Jr. weather station article (*Circuit Cellar* 78) and I was hooked. Of course, that was three years ago and the renovations have just begun, but that's another story.

One of my favorite things about the HCS-II is the simple yet powerful network protocol. It can carry lots of information but it's still easy to understand. Given the size of our house, I needed small HCS network nodes that I could locate almost anywhere. Although the Answer MAN Jr. fit the bill, in some applications it was overkill. And that's how the idea of designing my own HCS-II modules was born.

STARTING SIMPLE

One of the first things I wanted to add to the system was a printer to keep a log of events from my HCS-II, and Steve's idea of printing a daily weather report made it even more appealing. I decided to start my adventure with a tiny version of the DIO-Link.

An HCS-II network node interacts with the HCS-II while simultaneously dealing with the real world. The HCS-II constantly polls network modules for status information and sends commands when XPRESS needs something done. If a module does not respond to a query, the HCS flags it as offline.

What seemed like a simple undertaking was suddenly getting more complex. My network node had to handle a constant stream of network traffic and print text to the printer.

However, during the printing, the node still had to respond to queries from the HCS-II. Even though the HCS-II tries to be nice and not send two consecutive network packets to a specific module, it sometimes queries a module right after it sends a string to print.

SELECTING THE HARDWARE

Memory requirements were modest because the incoming packets would be no more than 96 bytes. The I/O requirements were 10 bits for the printer (8 data, STROBE, and BUSY), 3 bits for the serial interface, and 3 bits to set the node address (0–7).

To keep costs low, I was determined to implement this project with two chips: a processor and the 75176 RS-485 IC. Thanks to the wide variety of Microchip PICs, finding a suitable processor was easy.

I settled on the '16C63A because it had plenty of RAM for the serial buffer and 22 bits of I/O, which easily handled the I/O requirements. I even had extra bits to add blinky lights for network and I/O activity. My DIO-Link had become the PIC-DIO shown in Photo 1.

The great thing about the ’16C63A is the built-in serial UART, which made interfacing with the HCS-II RS-485 network even easier. The UART can store two incoming bytes and most of a third in its receive register before it overflows. Thus, my software can spend a lot less time worrying about incoming serial data.

#cc NODEX Command [Data]

#	Packet start character. # for checksum, ! for no checksum
cc	One byte checksum value in ASCII
NODEX	Module network address (i.e., DIO3)
Command	Module command (i.e., S DP= which sets the I/O port)
Data	This is optional, depending on the command

Figure 1—*The HCS-II uses a simple and flexible network packet structure. The checksum is optional based on the start character.*

TALKING WITH THE HCS-II

Figure 1 shows the general structure of an HCS-II network packet. Although the bit-level work is handled in hardware, the serial routine still has a lot to do. It has to manage the receive buffer, calculate and verify the checksum, compare the node address, and signal the main code when a packet is ready for processing.

Because the packet bytes are asynchronous, the serial routine is called via interrupt anytime a new byte is received from the network. This setup enables the module to concentrate on real-world interfacing between network bytes without using a multitasking OS.

Although I’m no stranger to assembler, I decided to use C for this project. I chose the CCS C compiler for PICs because it was inexpensive and had many powerful commands tailored to the PIC architecture. I have to admit, the final program was more complex than I thought it would be, so using C probably saved my sanity.

Listing 1 shows the serial routine that handles incoming HCS-II network traffic. It uses a simple 128-byte buffer implemented with the PIC’s FSR register for indirect memory addressing.

The C compiler has an array capability that V.1.0 of my code used for the buffer. However, the overhead required to manage a 128-byte array and the cycles needed to reference an array record caused me to manage the bank of RAM myself. The final code is quite efficient, thanks to the indirect address register.

The serial routine ignores incoming data until a packet-start character arrives. This character indicates whether a checksum is used. Checksums are optional for my module, although the HCS-II always uses them. The serial routine switches to receive mode and any subsequent characters are stored in the buffer, overwriting any old data.

The first version of this code stored all incoming packets in the buffer, regardless of the intended destination. Checksum and node address verification occurred after every packet was received, which wasted a lot of cycles.

In a simple digital I/O module like this, it wasn’t a big deal. But, for something like an LCD interface where a lot of processing can be required for each packet, these extra cycles were valuable.

When I rewrote the routine, I moved the address matching and most of the checksum calculating into the serial routine. The PIC-DIO now compares the address before the entire packet is received so it can stop storing data it will never process. The checksum calculations require quite a few cycles over an entire packet, so limiting it to packets that the module actually processes leaves more cycles for the main code.

The buffer setup is simple. Packets are stored in the buffer as they are received. Any data already in the buffer is overwritten. The effect is that the new data “chases” the unprocessed old data.

The PIC-DIO tracks the processing of the old data as new data is received behind it, and if the new data catches up, the packet is ignored. During normal operation, this shouldn’t happen because the PIC-DIO processes the parallel data much faster than the HCS-II sends data out.

Buffer overflows are handled as well. Because the checksum match fails on a partial packet, a buffer overflow causes the entire packet to be dumped.

This serial routine is used in many of my HCS-II projects. If I develop a module that takes a long time to process incoming data, I can improve the serial routine by moving to a circular buffer so that new data is stored after the old data. This provides more time before the new data can catch up with the old data.

IS IT VALID?

Once the whole packet is received, the checksum must be verified. HCS-II checksums are straightforward to validate. The two checksum characters are converted into a hex value and stored.

The checksum characters are replaced by zeros and all of the ASCII values in the packet up to, but not including, the carriage return are added together. Finally, the checksum value

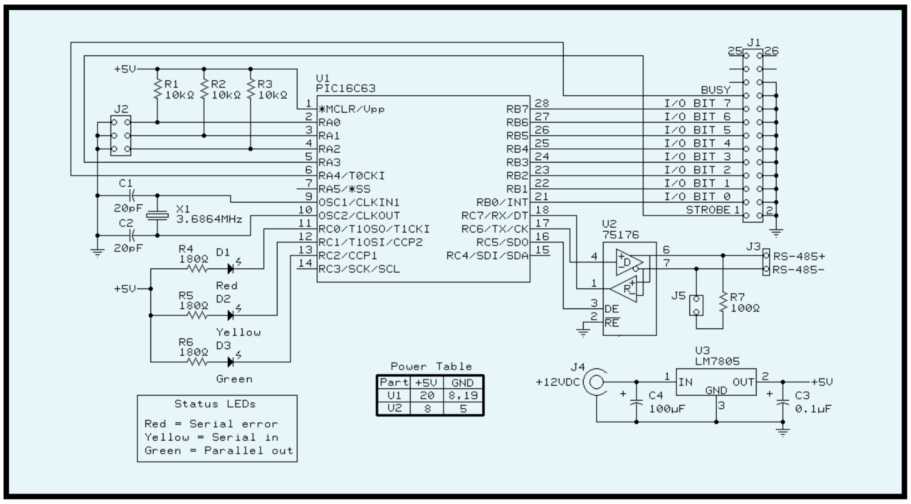


Figure 2—*The PIC-DIO hardware is quite simple. The node address is set using J2 while J5 enables network termination. J1 is wired for connection to a Centronics printer, but you can use the PIC-DIO for all types of I/O. Remember to add buffers if necessary!*

is added to the ASCII total. If the result is zero, the data is valid.

As you look at the code, you'll notice I took a shortcut. When you add up the first 10 bytes, they are always the same because the checksum bytes are always reset to zero.

I calculate the sum of the first 10 ASCII bytes during initialization, which enables the serial routine to only perform checksum calculations on data beyond the node address. Thus, the PIC-DIO only uses cycles to sum up the ASCII bytes if the packet is intended for it.

THE REST IS SIMPLE?

Now that we can receive HCS data, we have to do something with it. Considering all that the serial routine does, the command handling code is simple by comparison.

The processor sits in a `while` loop waiting for the serial routine to receive a complete, valid packet. When it does, the code breaks out and moves the processing buffer pointer up to the network command letter.

This process adds breathing room in the buffer as additional network data arrives. The code then uses a `switch` statement to execute the appropriate code based on the command letter.

The PIC-DIO recognizes four different commands from the HCS-II. The simplest one, `R`, initiates a module `reset` by timing out the PIC watchdog timer in an endless loop. The remaining three commands manipulate the I/O ports and are slightly more involved.

Although I designed the PIC-DIO with a printer interface in mind, I really wanted to design a tiny DIO-Link that would understand all of the DIO-Link commands. The HCS-II DIO-Link protocol enables you to set and query the I/O port bit by bit or as a complete byte. There's also a string command so a string of data can be output to the I/O port.

There is no command to set the direction of the I/O bits. By using pull-ups built into the PIC, you can set data and direction with one command (see Figure 2).

The PIC-DIO never outputs a high state. Instead, the PIC switches the port to an input and the pullup out-

Listing 1—*Even though the PIC-DIO uses a hardware UART, the serial input routine has a lot to do. It handles buffer management, checksum calculation, address matching, and error handling. The `serial_in` routine is called via interrupt when the PIC-DIO senses a start bit.*

```
void serial_in() {
  char data_in;
  int  tidx;
  // Read buffer until empty in case we have more than one char here
  while (bit_test(PIR1, 5)) {
  // Don't use getc; it does needless bit check to see if data is available
    restart_wdt();
    data_in = RCREG;
    // Use HW UART here for the sake of simplicity.
    // Let's act on the data... We are looking for start of a packet.
    // Otherwise stay in this state; ignore incoming char
    if (s_idle) {
      if ((data_in == '!') || (data_in == '#')) {
        // New Packet - Go to the front of the buffer
        LAN_IN = 0;
        serial_idx = 0;
        check_val = node_total;
        // Clear checksum buffer using default beginning
        write_buffer(0x00, data_in);
        s_idle = FALSE;
        S_ERR = 1;
        // Get correct offset for the address/node name
        if (data_in == '#') {
          tidx = 4;
          checksum = TRUE;
        } else {
          tidx = 2;
          checksum = FALSE;
        }
      }
    } else {
      // Check for buffer overflow or if we caught up to a previous packet
      if ((++serial_idx == BUFFER_SIZE) || (!p_idle && (serial_idx ==
        process_idx))) {
        // Buffer Overflow - dump it; checksum won't validate anyway
        s_idle = TRUE;
        S_ERR = 0;
        LAN_IN = 1;
      } else {
        // Check for end of packet
        if ((data_in == '\r') || (data_in == '\n')) {
          // Check checksum if we are supposed to
          if (checksum) {
            check_val += ((gethexbyte(read_buffer(1)) << 4) |
              gethexbyte(read_buffer(2)));
            if (check_val == 0) {
              go_data = TRUE;     // Tell main program to process buffer
            } else {
              // Bad checksum
              S_ERR = 0;
            }
          } else {
            go_data = TRUE;
          }
          s_idle = TRUE;  // If invalid checksum, we already have reset
          LAN_IN = 1;
        } else {
            // Save data
          write_buffer(serial_idx, data_in);
            // Check character against node address if necessary
          if ((serial_idx >= tidx) && (serial_idx < (tidx + 4))) {
            // It's an address character - check it
            if (thisnode[(serial_idx-tidx)] != data_in) {
            // Not our packet - break out and wait for next one
              s_idle = TRUE;
              LAN_IN = 1;
            }
          }
          if (checksum) {       // Add this into checksum value
            if (serial_idx >= 9) { // Add char to checksum value if needed
              check_val += data_in;
            }
} } } } } } }
```

Listing 2—*Incoming network data is processed in an endless `while()` loop using a `switch` statement. This listing shows all the code necessary to set bits on the I/O point using the `S` command.*

```
scratch = read_buffer(process_idx);   // Get command
switch(scratch) {
  case('R'):
    [Timeout Watchdog & reset PIC-DIO]
    break;
  case('S'):
    // Its a set command, bit or byte
    P_OUT = 0;
    process_idx += 2;  // skip to = or bit number

    if (read_buffer(process_idx) == '=') {
      // A byte set; easy! Convert ASCII hex into real hex value
      last_hcs_data = (gethexbyte(read_buffer(++process_idx)) *
        16) + gethexbyte(read_buffer(++process_idx));
      // Because we need open collector ports, use Tris command
      //   to set highs and normal port output for lows
      SET_TRIS_B(last_hcs_data); // Highs are high impedence
      portb = last_hcs_data;
      // Assert lows; highs already there due to pullups
      } else {
      // A bit set - little more complicated
      // Use last data sent by HCS to maintain proper tristate
      //   settings
      // We need this; a low input would reassert and flip tri
      //   causing possible short if we read port, flipped bit,
      //   and sent it back to port
      idx = 1;                // Prevent bogus bit numbers

      scratch = read_buffer(++process_idx);
      if ((scratch > '7') || (scratch < '0')) { break; }
      // Quick and dirty way to get 2^x

      for (scratch = (scratch & 0x0F); scratch != 0; scratch--) {
        idx <<= 1;            // Shift bit once
      }

      process_idx += 2;               // Shift to bit value
      if (read_buffer(process_idx) == '0') {
        // Twiddle bit using last HCS data so we won't grab
        //   inputs and reassert them
        last_hcs_data &= (idx ^ 0xFF);
        // Twiddle bit from HCS data, maintain inputs
        portb = last_hcs_data;
        // Set latches BEFORE changing TRIS
        SET_TRIS_B(last_hcs_data); // Assert low set in latches
        } else {
          last_hcs_data |= idx;          // Set specific bit
          SET_TRIS_B(last_hcs_data);
          // Pullups take care of setting port
      }
    }
    p_idle = TRUE;  // Release old buffer. We are done with it.
    // Pulse strobe on a port change in case user needs it
    delay_cycles(1);   // Let data stabilize
    strobe = 0;
    delay_us(250);     // Pulse strobe low in case they need it
    strobe = 1;
    break;
  case('Q'):
    [Read port data and return to HCS-II]
    break;
  case('T'):
    [Loop through string and output to parallel port]
    break;
}
```

puts a weak high. But if something external drives the pin, it can be read. Thus, if you set the port to 0xFF, you've done two things.

All pins go high unless driven externally and all pins can also be read as inputs. The only danger with this setup is if a pin is driven high externally and the HCS tells the PIC to drive the pin low.

The PIC-DIO bits can be set at the same time by sending the module a hex byte using `S DP=`*xx*, where *xx* is the hex byte to output on the I/O port. The HCS-II can also tell the PIC-DIO to manipulate a specific bit by sending the `S DP.#=`*x* command, where # is the bit number to manipulate and *x* is either a 0 or 1.

In this case, the PIC-DIO simply changes the appropriate bit and sets the whole port just like it does for a byte set command. Listing 2 outlines the code needed to set the I/O port pins based on the received data.

Photo 1—*The simple PIC-DIO hardware allows for quite a small circuit board. Imagine if you used surface-mount devices!*

READING THE PORT

The HCS-II constantly polls the PIC-DIO for the current state of the I/O port. The PIC-DIO returns the current state of each I/O pin as a single byte.

The HCS-II uses this reply as a way to tell whether a module is responding. If a module fails to reply to a query in a timely manner, the module is shown offline in the HCS-II host display.

When a query command comes in, the PIC-DIO reads the port and converts the byte into a two-character ASCII string to represent the hex value. The packet is constructed and the checksum is calculated as if the query packet had one. After a 50-ms delay to give the HCS-II a chance to switch to receive mode, the response packet is sent out via the UART.

Though the HCS-II doesn't use it, the PIC-DIO also accepts a bit query command so you can query the given state of a bit without getting the rest of the port data.

PRINTING TEXT

The final command enables the HCS-II to send a string to the PIC-DIO, which is sent to the I/O port. The PIC-DIO jumps to the first character of the string (after the `T DP=` command) and loops until the end of the string.

The HCS-II supports some escape sequences that must be checked for (see Table 1). Once the preprocessing is complete, the resulting data is sent to the printer.

Command	Definition
\n	New line (carriage return and line feed – ASCII 13, ASCII 10)
\r	Carriage return only (ASCII 13)
\xHH	Output ASCII code sent as hex byte
\f	Form feed (ASCII 12)
\e	Escape code (ASCII 27)
\t	Tab (ASCII 9)
\cX	Sent control code X (Ctrl-M = ASCII 13)
\\	Send \ character

Table 1—*The PIC-DIO understands the following escape commands when sending a string to the I/O port.*

If the BUSY line shows that the printer is busy, the PIC-DIO waits 25 ms for it to clear. If it doesn't clear, the current packet is dumped and the PIC-DIO waits for the next packet.

The PIC's RTCC timer is used to time the 25 ms. If the BUSY flag is clear, the data is sent to the I/O port. The STROBE line is pulsed for 25 µs and then the next character is processed. Although this setup was intended for a parallel printer, the STROBE line allows the PIC-DIO to be used as an interface to other circuits.

THE NEXT STEP

Once I got the code running, I went through the opcode listing manually to see how well some concepts were implemented. I didn't have to code any sections in raw assembler, but altering the C-code structure or changing the way some variables were referenced saved valuable resources.

The PIC-DIO provides a great platform for interfacing devices to the HCS-II RS-485 network. I hope you see now that interfacing to the HCS-II is a fairly easy task. Given the flexibility of the HCS-II serial protocol, the possibilities are endless. ■

Mike Baptiste works for Nortel Network's R&D facility in North Carolina's Research Triangle Park where he manages the Desktop and Intranet Services Support Groups. You may reach him at baptiste@cc-concepts.com.

SOFTWARE

The PIC-DIO firmware is available via the *Circuit Cellar* web site.

REFERENCES

DIO-Link manual, ftp.circuitcellar.com/CCINK/1992/Issue_28

Microchip PIC16C63A datasheet, www.microchip.com/10/Lit/PICmicro/16C6X/30605/index.htm

SOURCES

PIC16C63A
Microchip Technology, Inc.
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

PIC-DIO kits, HCS-II
Creative Control Concepts
(919) 304-3107
Fax: (919) 304-3107
www.cc-concepts.com

CCS PIC C compiler
Custom Computer Services, Inc.
(414) 797-0455 x35
Fax: (414) 797-0459
www.ccsinfo.com/picc.html

FEATURE ARTICLE

Jacob Apkarian

Internet Control

If you're looking for a systematic approach to designing and evaluating control system performance, check out these CAD tools. They take Jacob from making a mathematical model of his system right down to generating and implementing code.

The development of the fast PC, design software, and the affiliated Internet technology have significantly improved the design cycles in control system design and implementation.

In this article, I describe the various tools available that can take you from concept to real-time remote controller implementation, tuning, and monitoring within a few hours.

The example I use here is the 3DOF (degree of freedom) helicopter experiment shown in Figure 1. The 3DOF helicopter consists of a base on which a long arm is mounted. The arm carries the helicopter body on one end and a counterweight on the other.

The arm can tilt on an elevation axis as well as swivel on a vertical (travel) axis. Quadrature optical encoders mounted on these axes measure the elevation and travel of the arm. The helicopter body is mounted at the end of the arm.

The helicopter body is free to pitch about the pitch axis. The pitch angle is measured via a third encoder.

Two motors with propellers mounted on the helicopter body can generate a force proportional to the voltage applied to the motors. The force generated by the propellers causes the helicopter body to lift off the ground.

The purpose of the counterweight is to reduce the power requirements on the motors. The counterweight is adjusted such that the effective mass of the body is approximately 70 g.

All electrical signals to and from the arm are transmitted via a slipring with eight contacts. This setup eliminates the possibility of tangled wires and reduces the amount of friction and loading about the moving axes.

The purpose of the exercise is to design a controller that enables you to command the helicopter body to a desired elevation and a desired travel position. So, I want to describe a systematic approach to designing and evaluating control-system performance using available CAD tools.

MODEL DERIVATION

The first step in the process is to develop a mathematical model of the system. The equations that I reference in the text are displayed in the sidebar on pages 39–40 (they are also available for download in a pdf file).

The tool used here is the Maple symbolic processing language. The differential equations of the system are highly nonlinear and coupled, and are difficult to solve by hand. The process is described below and is implemented in a Maple script file.

Let's start by defining coordinate transformation frames embedded in the moving bodies of the system. The transformation matrices between the base frame and travel frame are shown in equation 1. These coordinate frames have the same origin.

Equation 2 shows the transformation matrices from the travel frame to the helicopter body. To go from the travel frame to the counterweight, the matrices are shown in equation 3. The last two sets of matrices are for the helicopter body to front motor (equation 4) and from the helicopter body to the back motor (equation 5).

The transformations from the base to the moving bodies are obtained via equation 6. Those transformations are used to obtain the potential energies of the bodies (equation 7) and the kinetic energy of each body (equation 8).

The total kinetic and potential energies are obtained and the Lagrangian

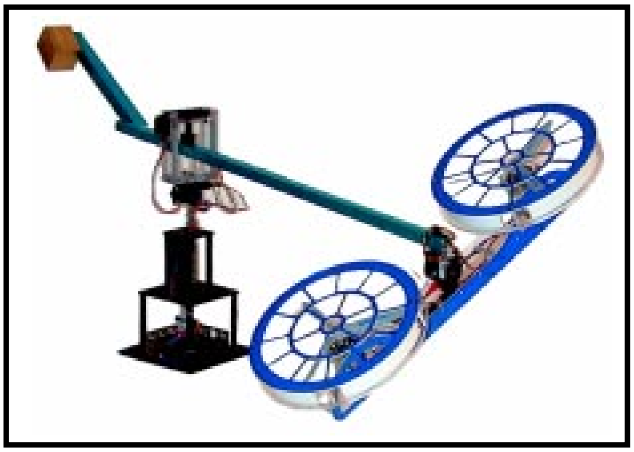


Figure 1—*The helicopter model consists of two motors driving two propellers mounted on a frame that can freely pitch. The frame is mounted on a long arm with a counterbalance. The entire arm can pivot and elevate, resulting in three degrees of freedom of movement.*

is computed in equation 9. Once the Lagrangians for the three axes are derived, you can write the Lagrange equations for each axis.

Each motor exerts a force normal to the body given by $F = K_fV$, where K_f is the force constant for the motor/propeller pair. If the body is horizontal, the two forces result in a torque about the elevation axis equal to La $K_f(V_f + V_b)$, resulting in the generalized Lagrange equation given in equation 10.

The system rotates around the travel axis only if the body is pitched and hovering. The torque around the travel axis resulting from a given pitch is equal to that component of the total force of the two motors projected onto the travel axis:

$$(V_f + V_b)\ K_{mLa} \sin(\text{pitch}(t))$$

resulting in equation 11.

The body pitches because of a difference in the voltage applied to the motors, which results in a difference in the two forces generated by the two motors. The difference results in a torque around the pitch axis given by:

$$K_f\ (V_f - V_b)_{Lh}$$

Equation 12 results in a set of nonlinear differential equations of the form given in equation 13. These are solved to obtain the accelerations around the three axes (equation 14), where $\varepsilon \equiv$ elevation, $p \equiv$ pitch, and $\lambda \equiv$ travel.

LINEARIZATION

These equations are then linearized around the operating point (equation 15). To derive the quiescent voltage, assume that the body is hovering so the force generated by the motors and the body weight should exactly counterbalance the counterweight. This equation can be written:

$$\text{Lw mc g} = \text{La}\ (g\ (m_f + m_b) + (V_f + V_b)\ K_f)$$

Assuming that both motors are exerting equal forces ($V_f = V_b$) and that both motors have the same mass ($m_f = m_b$), the body does not pitch, and we obtain equation 16.

Linearizing the acceleration equations around the quiescent point (Q) gives a set of equations of the form shown in equation 17. The elements of the A and B matrices are automatically derived and written into a MATLAB script from Maple.

CONTROL-SYSTEM DESIGN

Next, you can write a MATLAB script file that reads in the output file of the Maple program and calculates the values of the state space model A and B matrices for a given set of system parameters (equation 18).

In this design, I used a linear quadratic regulator (LQR) controller. The LQR method is essentially a multi-input/multioutput PID controller with an optimization index.

The optimization index requires two matrices, Q and R, which are used to compute a performance index to be minimized. The process is automatic.

You supply the Q and R matrices (which are selected intuitively), and the software computes the feedback gains. After several iterations using the simulation block (described next), we select a set of Q and R matrices that result in equation 19.

As you examine the feedback gains obtained from the LQR design, note that the second row gains have the exact magnitudes of the first row (see equation 20)! The state feedback equation is shown in equation 21. Further examination reveals that the sum of the two rows results in equation 22, which can be rewritten as equation 23.

Equation 23 is a PID controller around the elevation axis, which means that the gains we obtain from LQR design can be used in an elevation control loop (equation 24). Examining the difference between the gains (i.e., $V_f - V_b$) yields equation 25, which consists of two loops—one for pitch and one for travel.

This equation can be rewritten as equation 26, which is a PID loop to command the pitch to track the desired pitch (P_c). The desired pitch is defined in equation 27, which is another PID loop that controls the travel position. You now have the control equations shown in equation 28, and solving for V_f and V_b, you get the results in equation 29.

SIMULATION

Figure 2 shows the Simulink model used to simulate the system. It consists

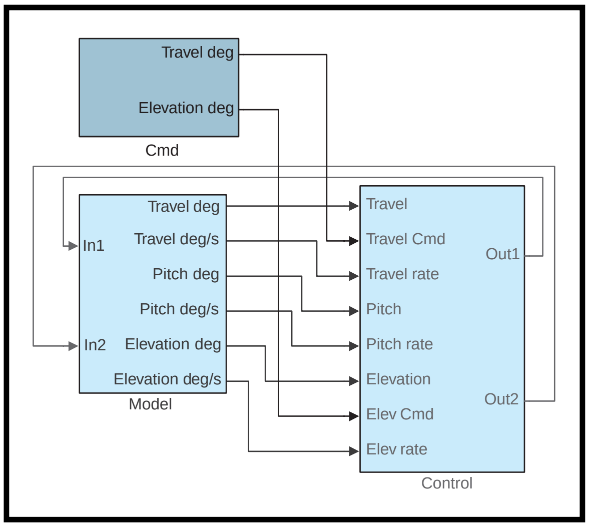


Figure 2—*The Simulink diagram consists of three main blocks: the command generation block, the open-loop model, and the controller.*

Equations

[1]

$$T_0^1 = \begin{bmatrix} \cos(\text{travel(t))} & \sin(\text{travel(t)} & 0 & 0 \\ \sin(\text{travel(t)} & \cos(\text{travel(t))} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[2]

$$T_1^{mc} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\text{elev(t))} & \sin(\text{elev(t))} & \text{Lw}\cos(\text{elev(t))} \\ 0 & \sin(\text{elev(t))} & \cos(\text{elev(t))} & \text{Lw}\sin(\text{elev(t))} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[3]

$$T_1^{hb} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\text{elev(t))} & \sin(\text{elev(t))} & \text{La}\cos(\text{elev(t))} \\ 0 & \sin(\text{elev(t))} & \cos(\text{elev(t))} & \text{La}\sin(\text{elev(t))} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[4]

$$T_{hb}^{f} = \begin{bmatrix} \cos(\text{pitch(t))} & 0 & \sin(\text{pitch(t))} & \text{Lh}\cos(\text{pitch(t))} \\ 0 & 1 & 0 & 0 \\ \sin(\text{pitch(t))} & 0 & \cos(\text{pitch(t))} & \text{Lh}\sin(\text{pitch(t))} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[5]

$$T_{hb}^{b} = \begin{bmatrix} \cos(\text{pitch(t))} & 0 & \sin(\text{pitch(t))} & \text{Lh}\cos(\text{pitch(t))} \\ 0 & 1 & 0 & 0 \\ \sin(\text{pitch(t))} & 0 & \cos(\text{pitch(t))} & \text{Lh}\sin(\text{pitch(t))} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[6]

$$\begin{aligned} T_{mc} &= T_0^1 T_1^{mc} \\ T_{hb} &= T_0^1 T_1^{hb} \\ T_f &= T_{hb} T_{hb}^{f} \\ T_b &= T_{hb} T_{hb}^{b} \end{aligned}$$

[7]

$$\begin{aligned} pe_{mc} &= mc\, g\, p_z^{mc} \\ pe_f &= mf\, g\, p_z^{f} \\ pe_b &= mf\, g\, p_z^{b} \end{aligned}$$

[8]

$$\begin{aligned} ke_{mc} &= 0.5\, mc\left(v_x^{mc\,2} + v_y^{mc\,2} + v_z^{mc\,2}\right) \\ ke_f &= 0.5\, mf\left(v_x^{f\,2} + v_y^{f\,2} + v_z^{f\,2}\right) \\ ke_b &= 0.5\, mb\left(v_x^{b\,2} + v_y^{b\,2} + v_z^{b\,2}\right) \end{aligned}$$

[9]

$$\begin{aligned} ke &= ke_{mc} + ke_f + ke_b \\ pe &= pe_{mc} + pe_f + pe_b \\ L &= ke - pe \end{aligned}$$

(continued)

Equations *(continued)*

[10]

$$\frac{\partial}{\partial t}\frac{\partial L}{\partial\left[\dot{elev}\right]} - \frac{\partial L}{\partial\left[elev\right]} = L_a Kf\left(V_f + V_b\right)$$

[11]

$$\frac{\partial}{\partial \mathrm{t}}\frac{\partial \mathrm{L}}{\partial\left[\dot{\mathrm{travel}}\right]} - \frac{\partial \mathrm{L}}{\partial\left[\mathrm{travel}\right]} = \left(\mathrm{V_f} + \mathrm{V_b}\right)\mathrm{Kf\,L_a}\sin\left(\mathrm{pitch}\left(\mathrm{t}\right)\right)$$

[12]

$$\frac{\partial}{\partial \mathrm{t}}\frac{\partial \mathrm{L}}{\partial\left[\dot{\mathrm{pitch}}\right]} - \frac{\partial \mathrm{L}}{\partial\left[\mathrm{pitch}\right]} = \mathrm{Kf}\left(\mathrm{V_f} - \mathrm{V_b}\right)\mathrm{L_h}$$

[13]

$$F_1\left(elev,\ pitch,\ travel\right) = G_1\left(V_f,\ V_b\right)$$
$$F_2\left(elev,\ pitch,\ travel\right) = G_2\left(V_f,\ V_b\right)$$
$$F_3\left(elev,\ pitch,\ travel\right) = G_3\left(V_f,\ V_b\right)$$

[14]

$$\dot{\varepsilon} = \mathrm{R}(\varepsilon,\ \mathrm{p},\ \lambda, \mathrm{V_f}, \mathrm{V_b})$$
$$\ddot{\mathrm{p}} = \mathrm{R}(\varepsilon,\ \mathrm{p},\ \lambda,\ \mathrm{V_f},\ \mathrm{V_b})$$
$$\ddot{\lambda} = \mathrm{R}(\varepsilon,\ \mathrm{p},\ \lambda,\ \mathrm{V_f},\ \mathrm{V_b})$$

[15]

$$\mathrm{Q} = \left[\varepsilon = 0,\ \mathrm{p} = 0,\ \lambda = 0,\ \dot{\varepsilon} = 0,\ \dot{\mathrm{p}} = 0,\ \dot{\lambda} = 0,\ \mathrm{V_f} = \mathrm{V_q},\ \mathrm{V_b} = \mathrm{V_q}\right]$$

[16]

$$\mathrm{V_q} = \frac{\mathrm{g(Lwmc - 2Lamf)}}{\mathrm{2Kf}}$$

[17]

$$\begin{bmatrix}\dot{\varepsilon}\\ \dot{p}\\ \dot{\lambda}\\ \ddot{\varepsilon}\\ \ddot{p}\\ \dot{\lambda}\\ \dot{\zeta}\\ \dot{\gamma}\end{bmatrix} = A\begin{bmatrix}\varepsilon\\ p\\ \lambda\\ \dot{\varepsilon}\\ \dot{p}\\ \dot{\lambda}\\ \zeta\\ \gamma\end{bmatrix} + B\begin{bmatrix}V_f\\ V_b\end{bmatrix}$$

[18]

$$A = \begin{bmatrix}
0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00\\
0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00\\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00\\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00\\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00\\
0.00 & 0.60 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00\\
1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00\\
0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00
\end{bmatrix}$$

$$B = \begin{bmatrix}
0.00 & 0.00\\
0.00 & 0.00\\
0.00 & 0.00\\
0.15 & 0.15\\
1.02 & 1.02\\
0.00 & 0.00\\
0.00 & 0.00\\
0.00 & 0.00
\end{bmatrix}$$

[19]

$$\mathrm{K} = \begin{bmatrix}
18.30 & 10.14 & -17.55 & 12.02 & 3.52 & -26.00 & 3.54 & -1.12\\
18.30 & -10.14 & 17.55 & 12.02 & -3.52 & 26.00 & 3.54 & 1.12
\end{bmatrix}$$

[20]

$$K = \begin{bmatrix}
k_{11} & k_{12} & -k_{13} & k_{14} & k_{15} & -k_{16} & k_{17} & -k_{18}\\
k_{11} & -k_{12} & k_{13} & k_{14} & -k_{15} & k_{16} & k_{17} & k
\end{bmatrix}$$

[21]

$$\begin{bmatrix}V_f\\ V_b\end{bmatrix} = -\begin{bmatrix}
k_{11} & k_{12} & -k_{13} & k_{14} & k_{15} & -k_{16} & k_{17} & -k_{18}\\
k_{11} & -k_{12} & k_{13} & k_{14} & -k_{15} & k_{16} & k_{17} & k_{18}
\end{bmatrix}\begin{bmatrix}\varepsilon\\ p\\ \lambda\\ \dot{\varepsilon}\\ \dot{p}\\ \dot{\lambda}\\ \zeta\\ \gamma\end{bmatrix}$$

[22]

$$V_f + V_b = V_s = -\left[2k_{11}\left(\varepsilon - \varepsilon_c\right) + 2k_{14}\dot{\varepsilon} + 2k_{17}\zeta\right]$$

[23]

$$V_s = K_{ep}\left(\varepsilon - \varepsilon_c\right) + K_{ed}\dot{\varepsilon} + K_{ei}\int\left(\varepsilon - \varepsilon_c\right)$$

[24]

$$K_s = \begin{bmatrix}2k_{11} & 0 & 0 & 2k_{14} & 0 & 0 & 2k_{17} & 0\end{bmatrix}$$

$$with$$
$$K_{ep} = -K_{s1}$$
$$K_{ed} = -K_{s4}$$
$$K_{ei} = -K_{s7}$$

[25]

$$V_f - V_b = V_d = -\left[-2k_{12}p - 2k_{15}\dot{p}\right] - \left[2k_{13}\left(\lambda - \lambda_c\right) + 2k_6\dot{\lambda} + 2k_{18}\zeta\right]$$

[26]

$$\mathrm{V_d} = \left[-2\mathrm{k}_{12}\left[\mathrm{p} - \frac{2\mathrm{k}_{13}\left(\lambda - \lambda_\mathrm{c}\right) + 2\mathrm{k}_{16}\dot{\lambda} + 2\mathrm{k}_{18}\zeta}{-2\mathrm{k}_{12}}\right]\right] - 2\mathrm{k}_{15}\dot{\mathrm{p}}$$

$$= \left[-2\mathrm{k}_{12}\,\mathrm{p} - \left[2\mathrm{k}_{13}\left(\lambda - \lambda_\mathrm{c}\right) + 2\mathrm{k}_{16}\dot{\lambda} + 2\mathrm{k}_{18}\zeta\right]\right] - 2\mathrm{k}_{15}\dot{\mathrm{p}}$$

$$= -2\mathrm{k}_{12}\left[\mathrm{p} - \mathrm{p_c}\right] - 2\mathrm{k}_{15}\dot{\mathrm{p}}$$

[27]

$$p_c = -\left[\frac{2k_{13}\left(\lambda - \lambda_c\right) + 2k_{16}\dot{\lambda} + 2k_{18}\zeta}{2k_{12}}\right]$$

[28]

$$\mathrm{p_c} = \mathrm{K_{tp}}\left(\lambda - \lambda_\mathrm{c}\right) + \mathrm{K_{td}}\dot{\lambda} + \mathrm{K_{ti}}\zeta$$
$$\mathrm{V_s} = \mathrm{K_{ep}}\left(\varepsilon - \varepsilon_\mathrm{c}\right) + \mathrm{K_{ed}}\dot{\varepsilon} + \mathrm{K_{ei}}\int\left(\varepsilon - \varepsilon_\mathrm{c}\right)$$
$$\mathrm{V_d} = \mathrm{K_{pp}}\left(\mathrm{p} - \mathrm{p_c}\right) + \mathrm{K_{pd}}\dot{\mathrm{p}}$$

[29]

$$\mathrm{V_f} = 0.5\left(\mathrm{V_s} + \mathrm{V_d}\right)$$
$$\mathrm{V_b} = 0.5\left(\mathrm{V_s} - \mathrm{V_d}\right)$$

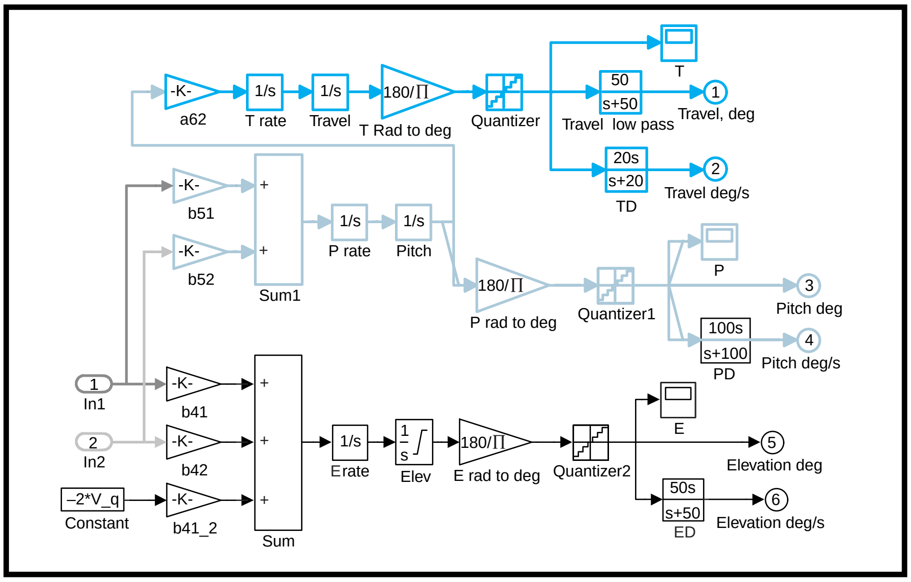


Figure 3—*In this open-loop model of the helicopter, the* A *and* B *matrices are reduced to a set of integrators. Quantizers simulate the quantization effect of the encoders. High-pass filters are used to obtain the derivative.*

of the open-loop model, command generation, and the controller.

Figure 3 shows the open-loop model, where system dynamics and measurement are simulated. The *A* and *B* matrices are reduced to a set of integrators and gains and the elevation integrator is limited to positive values to simulate the helicopter landing on the ground.

The gravitational bias is simulated by augmenting the motor voltages by the constants minus V_q. Encoder measurements are simulated by multiplying by a calibration constant that converts from radians to degrees. The quantization effect is simulated using a quantizer set to the resolution of the encoders.

The three displacement states are differentiated using high-pass filters, as they would be in the actual system. Numerical differentiation is not recommended, and high-pass filters function as differentiators at a frequency below the passband.

To evaluate the response of the system, you need to generate the commands to the elevation and travel axes. The command is for the elevation to rise to various levels and the travel to go +60° and return to zero.

Both commands are rate limited, using rate-limiting blocks. This block can, of course, be replaced by another input block (e.g., a joystick that lets the user command the system directly).

CODING

The controller is implemented using WinCon. This real-time Windows 95 application runs Simulink-generated code using Realtime Workshop to achieve digital real-time control on a PC equipped with a data acquisition and control board (DACB).

WinCon 3.0 consists of a client and a server. Each server can communicate with several clients. A PC can have a client as well as a server operating on it at the same time.

The WinCon server can convert a Simulink diagram to PC executable real-time code and run it in real time on a remote WinCon client PC. It also starts and stops this client remotely, as well as maintaining TCP/IP communications with several clients.

This server can also change parameters remotely, in real time and on-the-fly. These tasks are accomplished by using the Simulink diagram on the WinCon server PC or by using custom designed control panels on the WinCon server PC (without Simulink).

It also plots the data streamed back from the clients in real time on the server PC, saves collected data on a disk on the server PC, and runs script operations from the MATLAB environment to perform automated data collection, offline line adaptation,

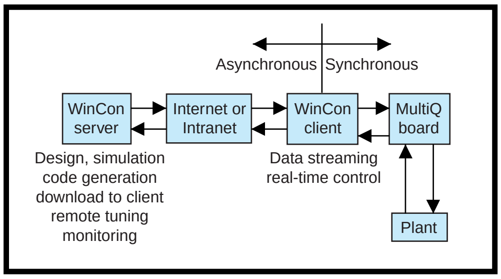


Figure 4—*In the WinCon Internet/Intranet configuration, the client runs in real time to control the plant via a data-acquisition board. The server, which runs remotely, enables the user to download the controller, tune it in real time, and monitor real-time data via the 'Net.*

automated design and parameter tuning, and so on.

WinCon Client is the real-time software component that runs the code generated from the Simulink diagram at the sampling rate you specify. It receives controller code from the server, runs the controller code in real time, maintains communications with a WinCon server and updates parameters in real time, and streams real-time data to the WinCon server requesting it.

The performance of this real-time component depends on several factors including the processor speed, controller complexity, and the number of other processes simultaneously running on the same PC. Testing revealed a maximum latency of 50 µs when the controller was run on a P200 with no other programs running.

CONFIGURATIONS

The simplest configuration is a single PC equipped with the appropriate software and hardware but with no network. In this configuration, the PC runs both the server and the client and can be used to perform real-time control, tuning, and monitoring in the same location.

The second configuration consists of two PCs—one running the server and another running the client. In this case, the two PCs must be connected via Ethernet.

The advantage of this configuration is that the client is running on a PC that's usually not running any software other than Windows 95 and WinCon W95Client. This setup gives you the fastest possible sampling rates because the control PC isn't burdened with other tasks.

The third configuration consists of two PCs communicating via the Internet. Each PC is connected to a server, and they can each be located anywhere in the world. Essentially, this configuration the same as the second one, but the connection between the PCs is via the Internet (see Figure 4).

The last configuration is one server and many clients running as nodes on the Internet. The server can download code to several clients and can maintain communications with all clients simultaneously.

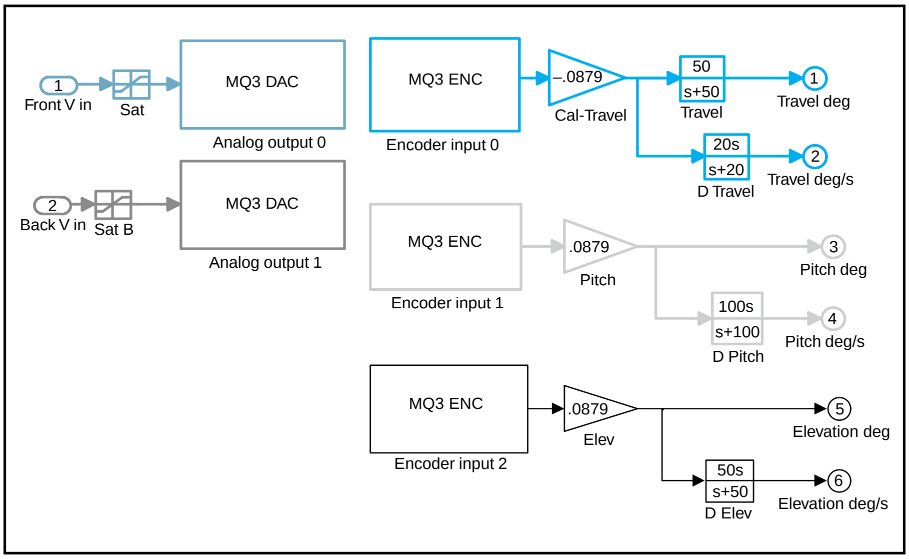


Figure 5—*In this actual-system inputs and outputs subsystem, the data-acquisition blocks represent measurements from encoders and outputs to D/A channels.*

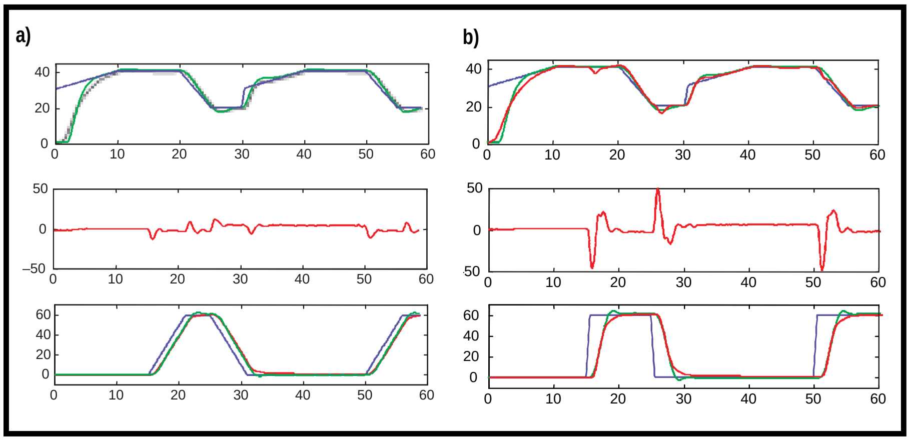


Figure 6a—*This graph shows the results with the pitch limit set to 15° and the travel rate limit set to 15°/s. The top plot is desired (blue), actual (red), and simulated (green) elevation. The second graph is actual pitch in degrees. The third graph is desired (blue), actual (red), and simulated (green) travel. Note how close the actual and simulation results are.* ***b***—*This graph shows the results with pitch limit set to 45° and travel rate limit set to 100°/s. The top plot is desired (blue), actual (red), and simulated (green) elevation. The second graph is actual pitch in degrees, and the third graph is desired (blue), actual (red), and simulated (green) travel.*

IMPLEMENTATION

To run the controller in real time, simply replace the simulation block in the simulation with the block in Figure 5. The difference between the simulation block in Figure 4 and the actual system block illustrated in Figure 5 is that the voltages coming back from the controller are connected to two analog output blocks.

The analog output blocks output the desired voltage to the MultiQ analog output channels 0 and 1. Furthermore, the three variables are not simulated. They are measured directly using the MultiQ Encoder input blocks.

Next, you generate C code from the diagram and compile the diagram using Realtime Workshop and the Visual C++ compiler. This step generates a library file that can be downloaded into the client.

TUNING AND RESULTS

My system is implemented as described in the third configuration. The server is located in the engineer's office, and the client computer and helicopter are in the laboratory, one floor above.

Note that the operator of the server can't even see the helicopter while running the controller. He has to rely solely on the server's real-time plotting capabilities to monitor the system's performance.

I augmented the block diagram to include the simulation of the system here. Therefore, I needed to generate code that runs in real time and performs two tasks—running the system and simulating it at the same time.

Using this technique, I can simultaneously monitor the simulation and the actual response, which is not usually done because it consumes processor power. However, I'm sampling only at 100 Hz, and the computations required for the simulation won't lengthen the computation time significantly.

Figure 6a shows the results. The simulation and actual results match quite closely, indicating that the linear model is a good representation of the system under these conditions.

But note that the pitch command is limited to 15° and that I limited the travel rate command to 15°/s, operating close to the linear region. What happens if you let the helicopter pitch to 45° and tell the system to travel at 100°/s?

This question is easily answered by changing the associated parameters in the relevant block and running the simulation and real-time controller again. The results are shown in Figure 6b. Note that although the simulation and the actual system match relatively closely this time, they diverge when the helicopter pitches to 45°.

They diverge because the simulation doesn't take into account the fact that a pitch reduces the effective vertical thrust. This behavior shows that care must be taken when simulating complex systems with nonlinearities.

Real-time tuning can be performed by changing the values right off the diagram or by implementing a WinCon control panel. This panel enables you to associate controller parameters with sliders, knobs, or switches and gives you access to these parameters independent of the Simulink diagram.

READY FOR TAKEOFF

In this article, I described a systematic approach to computer-aided design and rapid prototyping of digital feedback controllers. As you've seen, it's possible to quickly develop nonlinear and linear models using Maple and design controllers using MATLAB.

Simulations are implemented using Simulink and real-time controllers are implemented using Realtime Workshop and WinCon. WinCon's Internet capabilities facilitate the remote control and monitoring capabilities, enabling the user to test designs remotely, from across the room or across the globe.

Jacob Apkarian is the president of Quanser Consulting, which designs and manufactures advanced control system experiments to teach control systems. You may reach Jacob at jacob@quanser.com.

SOFTWARE

Source code for this article is available via the *Circuit Cellar* web site. Also, all equations mentioned by number are available for download in a pdf file.

SOURCES

WinCon, **MultiQ**
Quanser Consulting, Inc.
(905) 527-5208
Fax: (905) 570-1906
www.quanser.com

MATLAB, **Simulink**
The Mathworks, Inc.
(508) 647-7000
www.mathworks.com

MAPLE
Waterloo-Maple, Inc.
(800) 267-6583
(519) 747-2373
Fax: (519) 747-5284
www.maplesoft.com

RECONFIGURABLE SBC

MiroTech has introduced **ARIX**, the industry's first single-board reconfigurable computer (SBRC). The full-size scalable PCI board supports a TMS320C44 DSP coupled to a Xilinx Virtex 300, 800, or 1000 FPGA and outranks multiprocessor architectures of more than 50 DSPs from the same family.

The architecture is expandable through mezzanine sites that support its TIM standard, as well as the IndustryPac standard. Additionally, the ARIX architecture has an onboard topology of virtual processing elements that can support up to the equivalent of one million reconfigurable gates. They can be linked to other SBRCs through a series of external connectors, enabling the user to create a cluster of reconfigurable computers. High-speed (100 MHz) SBSRAM banks of 32 × 128 Kwords provide ample memory for the networked VPEs as well as for the DSP.

Because it supports Windows, ARIX allows end users to develop and partition their C application in hardware and software threads.

ARIX sells for **$7350**. The package includes a platform, the run-time environment, development and debugging tools, and compiler and libraries of hardware cores.

MiroTech Microsystems, Inc.
(514) 744-6476
Fax: (514) 744-6018
www.mirotech.com

Y2K BOARD

The parvus **Y2K Board** is a low-cost add-in with a secondary real-time system clock that correctly handles the century changeover in hardware. By reading this secondary clock directly from an application, Y2K-related errors are eliminated.

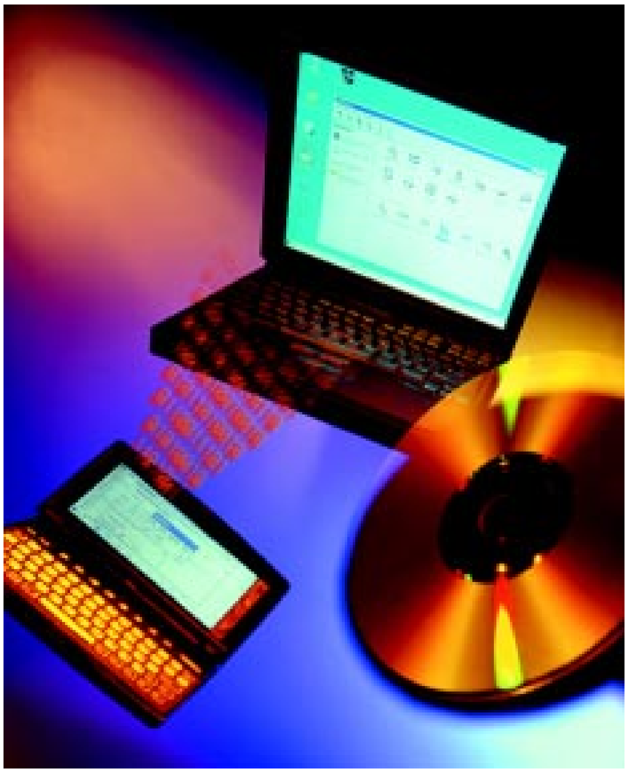
The Y2K Board works on any 'x86 platform, regardless of the operating system, other software, or any drivers that may be installed. It is available in standard ISA-bus or PC/104 format.

The board is fully compatible with the industry-standard real-time clock at address 70-71. It features I/O address selection, optional EPROM select, a periodic interrupt, and a socket for 27C256 EPROM. A hardware switch is included to eliminate the possibility of a rogue piece of software changing the date and time.

Software routines are provided to detect the board, set date and time directly, copy date and time from system, and read date and time string.

The Y2K Board includes a manual and diskette containing software drivers and source code examples. It is priced at **$99** for the ISA-bus and **$129** for the PC/104 version. A Y2K Board Development Kit, with disk and source code, is also available.

parvus Corp.
(801) 483-1533
Fax: (801) 483-1523
www.parvus.com



INFRARED UTILITY SOFTWARE

IrDirector is a software utility that enables computer users to quickly and easily configure and manage infrared cordless data connections. It automatically sets up connections for cordless connectivity, such as between an IR-enabled notebook computer and an IR-enabled electronic organizer, and provides users with a display that shows the connection status. Applications include IR-enabled notebooks and desktop computers, PDAs, digital cameras, cell phones, scanners, and other electronic systems.

IrDirector initially configures the user's PC for IR connectivity and addresses port conflict problems. It then loads drivers for newly discovered IR-enabled appliances. IrDirector simplifies infrared connectivity by displaying which applications are associated with each IR-enabled device and automatically launches it on connection. IrDirector's various components can be updated via the Internet, and an icon in the Windows "tray" alerts users to the status of the IR communications subsystem.

IrDirector sells for **$19.95** with significant discounts for OEM quantities.

Calibre, Inc.
(408) 573-3890
Fax: (408) 573-3899
www.calibre-inc.com

100-Mbps PCI ETHERNET

The **MiniModule/ESB** provides a PCI Ethernet interface, two serial ports, and a solid-state disk (SSD) expansion socket on a PC/104-*plus*–compliant module. The Ethernet subsystem is based on a PCI-interfaced AM79C972 controller, and it supports both 10- and 100-Mbps full-duplex data transfer.

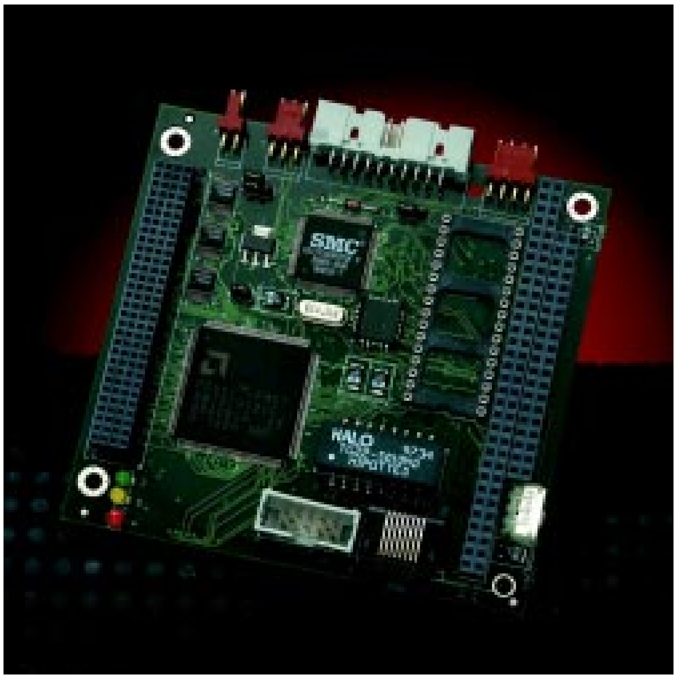
As well, 10BaseT or 100BaseT twisted-pair media connection is achieved through an onboard industry-standard RJ-45 connector. The Ethernet controller is supported by a wide range of operating systems including DOS, Windows 98, Windows NT, Windows CE, QNX, VxWorks, and other leading RTOSs.

The PC-compatible serial ports are implemented with 16C550-type UARTs that have 16-byte FIFO buffers for fast throughput. One serial port can be configured to support either RS-232 or RS-485 operation; the other serial port is configurable for RS-232C or TTL-level signaling.

A bytewide socket allows plug-in addition of an M-Systems DiskOnChip2000 solid-state disk. And, it too supports a wide range of operating systems.

The MiniModule/ESB module is priced at **$169** in OEM quantities of 100.

Ampro Computers, Inc.
(408) 360-0200
Fax: (408) 360-0222
www.ampro.com



Nouveau PC

Real-Time PC

Ingo Cyliax

Where in the World...

Part 2: Data Collection in Flight

Ingo's ready to get his application off the ground. No kidding, he really is headed for the wild blue yonder as he uses a PC/104 module to build a data-collection device so an ultralight aircraft can record topographical images.

As you probably figured out, I didn't just decide to write about GPS for no reason at all. I'm actually working on a neat GPS application. The mission: build a data-collection device for use in an ultralight aircraft.

The device I'm building has to perform several functions. Its primary function is to collect images from a downward-pointing camera that are correlated with GPS data. The device also helps the pilot by performing navigation functions that keep the ultralight on-course for the data runs. It may also be called on to collect ancillary analog data, also correlated with the GPS positional data.

After the images and data are collected, they are downloaded to a laptop computer when the plane is on the ground for refueling. Finally, the raw data is later processed to make maps. For example, by using infrared images, vegetation can be classified (weeds vs. crops).

Of course, there's much to write about here. Besides the GPS subsystem, which I cover this month, there is a touch-panel display and a video frame grabber. Also, because this system rides in an airplane, there are a number of environmental demands. Be on the lookout for future articles.

But first let's recap some of last month's ideas.

Photo 1—*Here's the engineer in back (EIB) verifying the operational requirements for the data acquisition system. The round thing behind me in the fuselage is the ballistic parachute used to land the plane in case the wing falls off.*

GPS REVIEW AND APPLICATIONS

The Global Positioning System (GPS) is used to compute positions and time anywhere in the world. Currently, there is no licensing fee for end users. You simply buy a GPS receiver, plug it in, turn on the power, and it works.

Basically, the GPS is a precise time-measuring system. It is so accurate that we can compute the time-of-flight delays. It's only natural that we can use the system to synchronize time. Imagine this; you can synchronize clocks anywhere in the world (or in space) to better than 100 µs. With special time receivers, even accuracies of under 100 ns are possible.

One application for such accuracy is to synchronize networks. In fact, many organizations on the Internet use GPS receivers to synchronize the real-time clock on computers and servers. Having synchronized time is important for applications such as key exchanges for encryption algorithms and keeping the data of distributed databases and file servers consistent.

Another application is to synchronize radio communication systems. This reduces the channel bandwidth because you don't have to send clocking information over the communication channel.

Photo 2—Here you see the computer box, the battery, display, and external GPS receiver. The computer is a ruggedized EBX module with a PC/104 stack. The display is a daylight-readable 8.4″ LCD flat-panel with touch-sensitive screen.

Both the transmitter and receiver can receive and decode GPS signals to arrive at a clock to time the data. You can think of the GPS as a big global synchronous clock. Many time-division multiplexing systems, such as those used in the satellite and telecom industries, rely on accurate system-wide timing.

Radio astronomers use radio telescope arrays with long baselines to map radio sources in the sky with high accuracy. They point radio telescopes at the same region in the sky and record the signal to a tape along with an accurate time signal. The tapes from different telescopes are then correlated.

Before GPS, radio telescopes required expensive atomic clocks at each site that were painstakingly synchronized. With GPS, it's possible to use less accurate clocks and synchronize them with GPS.

Radio telescopes are one example where we use GPS to aid in data collection. Other data-acquisition applications can use GPS to annotate data with positional information.

For example, GPS can be used by an engine controller in a commercial truck to collect data about fuel consumption. You'd expect fuel consumption to be higher in mountainous areas, so when you analyze the data, you can consult a database to see whether the fuel consumption is normal for the terrain the truck was driving in.

GPS can also be used to control automatic machinery. In large mines, big dump-trucks transport dirt and ore in difficult and dangerous terrain. Differential GPS enables trucks and other machinery to traverse the area more consistently than human operators, much like an autopilot. Although this might seem pretty futuristic, some companies are already testing this concept.

BUILDING A GPS DATA LOGGER

As I mentioned earlier, I'm building an airborne data logger. The airplane—really the ultralight aircraft shown in Photo 1—flies a grid pattern and takes pictures with regular and infrared cameras. The images are tagged and stored on a (big) hard disk along with the GPS data that was computed when the image was taken. Besides this, I want the GPS to provide basic navigation functions to the pilot.

Let's look at the system hardware first. Then I'll tell you about the user interface and its issues.

The system consists of an EBX form-factor CPU module and a PC/104 stack for I/O. PC/104 enables us to build rugged systems, and many PC/104 modules are available in extended temperature specifications.

Ultralight cockpits are not environmentally controlled. The system goes from whatever temperature and humidity conditions are on the ground, to conditions at operating altitude in a fairly short time (tens of minutes). The extended temperature specifications are a requirement here.

Ultralights also vibrate a lot, and the system has to withstand an occasional hard bump during landing. The PC/104 form factor gives me some assurance that the boards won't fall out.

Besides, all of the connectors used internally and externally must have a locking mechanism so they don't slip off. If it's not possible to lock them with a clip or screw, they have to be tie-wrapped together.

The hard disk I use is a 2.5″ laptop drive. These drives are designed to withstand quite a bit of shock while operating, but they still need to be mounted in a way that mechanically isolates them from the outside as much as possible.

This setup can be achieved with special shock-absorbing mounts or by wrapping the drive in shock-absorbing foam. If you wrap it in foam, make sure enough surface area on the hard disk enclosure is exposed so that it does not overheat.

Hard disks are the weakest link in this kind of system. Solid-state disks are the media of choice in this application. However, because we have to collect images, the costs of a sufficiently large solid-state disk would be astronomical.

Consider this: A typical flight lasts about 2 h. Each video image of 640 × 480 × 24 bits is about 900 KB. Compression depends on the image's complexity and the compression algorithm used.

Assume we get about 50% compression without image quality loss. A typical frame (f) rate might be as quick as 10 s/f.

$$\begin{aligned} \text{space} &= 500\ \text{KB/f} \times 2\ \text{h} \times 3600\ \text{s/h} \times 0.1\ \text{f/s} \\ &= 351\ \text{MB} \end{aligned}$$

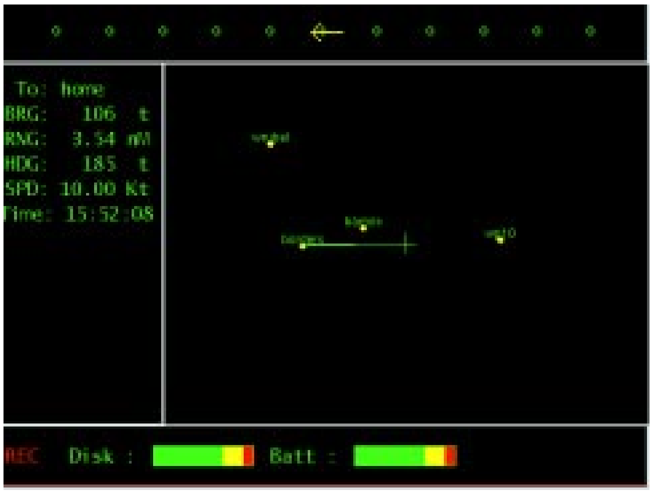

```
To:  home
BRG:    106  t
RNG:   3.54 nM
HDG:    185  t
SPD:  10.00 Kt
Time: 15:52:08

REC   Disk :        Batt :
```

Photo 3—To the left in this GUI is pertinent navigation information (e.g., heading, speed and bearing, distance to the next waypoint). At the top is the course deviation indicator. The needle indicates the direction to turn, and the horizontal mark is the cross-track error. Keep the needle straight and in the middle, and you'll fly right over the destination waypoint.

All of the components are mounted into a case (see Photo 2). For the user interface, an 8.4″ daylight-readable LCD with resistive touch panel can be mounted in the front, either on the instrument panel or the control yoke. It can also be operated on the lap. The computer with all of its I/O and batteries is placed out of the way, behind the pilot.

To make interfacing the panel easy, I use standard analog VGA from the computer. The panel has a VGA–to–flat-panel adapter that converts the analog VGA signal to the digital signals necessary to drive the LCD panel. The touch panel uses an RS-232 serial interface, much like a mouse to communicate with the computer.

Ultralights generally don't have an alternator, so for power I use a 12-V sealed lead-acid battery. You can find 12-V batteries everywhere, and compared to NiCd or more advanced batteries, they're easy to charge and deal with. They're also inexpensive.

The computer uses a 25-W DC/DC converter which operates from 9- to 18-VDC input and provides both 5- and 12-V outputs. The panel is powered directly from the 12-V battery because it has its own voltage regulators and inverter for the backlight. The video camera and GPS receiver also operate directly from the 12-V battery.

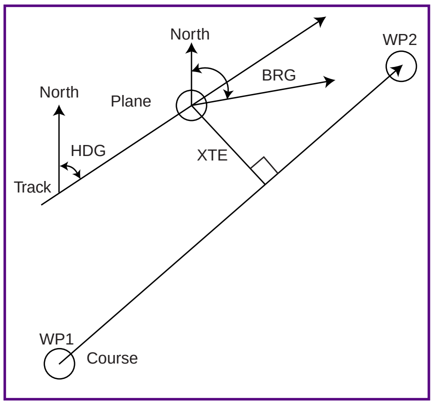


Figure 1—In the basic navigation model, the track is the path that the plane is flying, and the course describes the great circle route from the active "from" waypoint to the active "to" waypoint. The heading (HDG) is the direction the plane is currently traveling, and the bearing (BRG) is the direction from the plane to the "to" waypoint. The cross-track error (XTE) is the distance between the current location on the track and the course. The object is to make the HDG equal to the BRG and minimize the XTE.

The system is designed to use an external GPS receiver that plugs into one of the serial boards on the CPU module. The simple serial interface consists of transmit, receive, and ground. Ready-made serial cables can be bought from the GPS manufacturer and plugged directly into a PC-compatible serial port.

As I mentioned in Part 1, most GPS receivers output information using a standard protocol called NMEA-0183. The communication rate is usually 4800N2 but can be configured to most GPS receivers. Let's move on to look at the software.

This system runs Linux. I chose Linux because it's relatively easy to obtain and install. It is also multiprocessing and multithreaded, and it provides memory protection for processes.

Linux has a real-time extension in case I need tighter control over latency and interrupt response. My prototype doesn't need to be very tight in terms of latency, but it may be in the future if I need to add support for high-speed data acquisition and/or using high-tolerance timing signals.

Linux also supports almost all of the typical hardware found in Intel platforms, like IDE drives, CD-ROMs, and some types of flash-memory disks. Also, I can include the sources to everything on the disk, if I needed to make fixes in the field.

It's easier to send someone a patch to a source file than it is to get a binary upgrade or patch to them. You can always send a source-level patch as a printout over a fax or talk someone through making the changes on the phone. This capability is particularly important in remote locations where most of the remote sensing takes place and where downloading a binary file is just not that easy.

Check some of my earlier articles on embedding Linux to see what it's all about ("Embedded RT-Linux," *Circuit Cellar* 100–104). I want to concentrate here on making GPS work under Linux for my application.

I use Tcl (Task Control Language) for much of the code. Follow along by looking at Listing 1, which is code that collects GPS information.

To access the GPS receiver under Linux, open the special file `/dev/gps`, which I linked to the serial port device (typically `/dev/ttyS0` for the first serial port [COM1 in DOSese]). By making symbolic links to the actual physical device entries, it's easy to switch the function to different ports.

In any case, under Linux we open the serial port on the line that looks like:

```
set gpsfds [open "/dev/gps" "r"]
```

Tcl programs can be written in event-driven style. For this, use the `fileevent` call register and the file descriptor `gpsfd` to call `gpsread` whenever it is readable.

```
fileevent $gpsfd read-
   able gpsread
```

At the end of the initialization code, call the event handler `vwait`. Writing event-driven code is a nice practice, since programs that are written in the graphics dialect of Tcl (Tcl/Tk) are also event driven. For the sample code, however, we can live with a non-GUI-based program.

As I mentioned, the `gpsread` procedure is called whenever data is available on the file description. The default behavior of this file interface is to buffer up all the data until an end-of-line occurs.

That's fine for this application because NMEA sentences are formatted as lines of data. This way, when `gpsread` is called, an NMEA sentence is available to read.

```
while { [gets $gpsfd line] !=
   -1 } { ... }
```

The `while` loop reads NMEA sentences until the buffer is exhausted and then returns to the event handler.

After a sentence buffer is collected, we strip off all the leading characters. That's all the characters until we see a beginning of an NMEA sentence `$`.

Tcl has some nice text-processing features. For example, the call to `split` in `gpsread` parses the comma-separated NMEA sentence into individual fields.

Because we're only interested in position and the fix time, we look for NMEA sentences that have this information in them—here, the `GPRMC` sentence. As you recall from last month, `GP` stands for GPS source, and `RMC` for recommended minimum specific to GPS/transit data. We simply test for the string `$GPRMC` in the first field of the sentence and write it to the `gps`

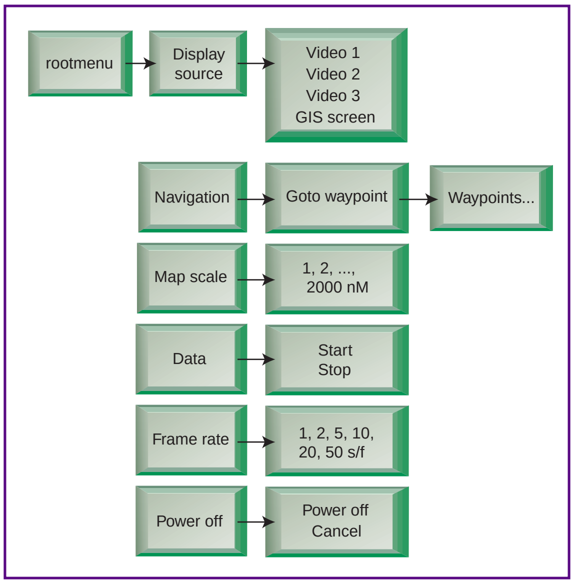


Figure 2—This menu system is designed to be easy to use. The pilot simply taps the display and the root menu appears. Most commonly used functions are accessible from there.

Listing 1—This is the code for basic GPS interface on a serial port. The code is written in Tcl (Task Control Language).

```
#! /usr/bin/tclsh

set picrate    10
set done ""

set gpsfd [open /dev/gps "r+"]
set logfd [open gps.dat "w"]
fconfigure $gpsfd -mode 4800,n,8,2
exec stty 4800 < /dev/gps

proc readgps { } {
  global gpsfd logfd
  while {[gets $gpsfd line] != -1} {
    if {$line == "ASTRAL"} {
      puts "Sync"
      puts $gpsfd "ASTRAL" ; flush $gpsfd
    }
    while { [string length $line] > 0 } {
      set time [clock seconds]
      if { [string index $line 0] == "\$" } {
        set fields [split $line ","]
        if { [lindex $fields 0] == "\$GPRMC" } {
          puts "$time $fields" ; flush stdout
          puts $logfd "$time $fields" ; flush $logfd
        }
        break;
      }
      set line [string range $line 1 end]
    }
  }
}
fileevent $gpsfd readable readgps
vwait done
```

data file indicated with the `gpsfd` file descriptor. This file descriptor was opened to write to the ASCII file `gps.dat`.

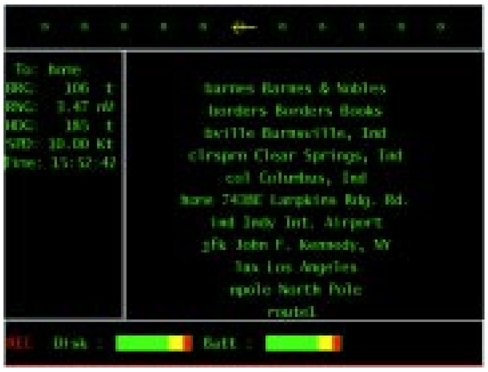
The Tcl routine `clock seconds` is called to get a unique marker that makes it easy to associate data collected in different parts of the system with the GPS data. The reason behind this is simple: the GPS fix time is obviously the most accurate, but it was calculated in the GPS receiver before the NMEA string was sent. By the time our system gets the string and passes it to the software's upper layer, time has passed.

Other code sections collect data as well, but they don't have access to the GPS data we just collected. The key enables us to associate the data in postprocessing. If we can determine the latency from when the fix was taken to when the key was generated, we can later correct the position. The best place for timestamping the GPS data is in the interrupt service routine for the serial port; probably the time that the first character of the sentence is received.

Timestamping is important if we collect data in a fast moving aircraft. At 500 knots, we cover about 825 ft./s (250m/s), so our position will be off quite a bit unless we can calculate the exact time the fix was obtained.

NAVIGATION

I talked to pilots who do this kind of work to find out some of their user-interface requirements. I also rode in one of the planes to get a firsthand view of the cockpit environment. Well, someone has to do all the dirty work….



```
To: home
BRG:   106  t
RNG:  3.47 nM
HDG:   185  t
SPD: 10.00 Kt
Time: 15:52:47

barnes Barnes & Nobles
borders Borders Books
bville Barnsville, Ind
clrspgn Clear Springs, Ind
col Columbus, Ind
home 7438E Lampkins Rdg. Rd.
ind Indy Int. Airport
jfk John F. Kennedy, NY
lax Los Angeles
npole North Pole
route1

REC  Disk :   Batt :
```

Photo 4—This menu on the GUI lets the pilot choose the next waypoint to travel to. I made sure that some of my favorite coffee shops were included so I can find them in fog or rain.

The interface should provide information in an uncluttered way and have the most common functions available quickly. Also, the navigation information needs to be in a format pilots are used to. Keep in mind that audible alarms and feedback are useless, due to the noise levels.

Because cockpits are crowded, pilots want one display that provides the functionality of their aircraft GPS receiver in addition to the data-collection function. Fortunately, we have all the necessary data.

Pilots need to know where the plane is headed (heading), the speed over the ground, and the elevation. They also want the computer to figure the distance and bearing to waypoints (i.e., navigational landmarks that the pilot wants to fly to for each leg of a flight).

Our pilot programs in waypoints for a grid pattern he wants to fly to collect data. Figure 1 shows the navigational model. Photo 3 shows a typical navigation display, including a moving map display.

A course deviation indicator (CDI) combines several pieces of information graphically. It is a needle that shows which direction the pilot must turn to get to the next waypoint (see top of Photo 3). The needle moves on a horizontal scale, which indicates how far off the desired course the pilot is. The error is called XTE (cross-track error) and is labeled in Figure 1.

The moving map display shows a situational display of the surrounding area. The center of the map is the location of the plane, while the map moves in a layer below.

At a minimum, this display shows the nearest waypoints and the airplane's track. Normally, north (magnetic or true) is at the top of the map. Some maps can be configured to rotate, so the top of the map is the current heading. A future version of this user interface will let you superimpose scanned maps on the moving map display.

A thermometer display shows the current state of the data recorder, disk space, and estimated battery capacity. If we're recording, a red "REC" legend is also shown.

In flight, the pilot wants to control the selection of waypoints and the frame rate of the imaging system. The frame rate depends on the altitude above ground, the speed, and the camera's field of view. The best frame rate allows some overlap of the picture, to give continuity, but not too much overlap, which wastes disk space.

The controls are done with pop-up menus. This way, all of the screen area is usable, unless the pilot wants to interact with the system. I organized the pop-up menu by functional groups (see Figure 2).

To activate the menus, the pilot taps the screen and the root menu appears. He then selects the submenus and functions or makes the menu go away by tapping outside the menu area. Photo 4 shows the screen after the "Goto Waypoint" submenu is selected.

Oops, out of room already! Next month, I'll point the way toward more GPS application information. RPC.EPC

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.*

SOURCES

Flat-panel displays
Earth Computer Technologies, Inc.
(949) 361-2333
Fax: (949) 361-2121
www.flat-panel.com

PC/104 system
VersaLogic Corp.
(800) 824-31163
Fax: (541) 485-5712
www.versalogic.com

GPS receivers
Garmin Corp.
(800) 800-1020
(913) 397-8200
www.garmin.com

Applied PCs

Fred Eady

Data Serving via the Internet

Part 2: Forms

Forms are the key to making the Internet interface, but designing good forms can be the key to frustration. With EmWeb archives and HTML, Fred shows us how we too can create simple and effective web forms.

Remember when I was grumbling about having to use Bill's year-named products to get screen captures from a third-party graphics program? I finally got so tired of making excuses, I asked the *Circuit Cellar* readership for help. Ask and you shall receive.

Thanks to all of you that took the time to clue me in on using Bill's non-year-named OS to get my screenshots out to the masses. In my eyes, it's just another example of the caliber of *Circuit Cellar's* readers.

Now that I have almost six months before my no-year-named OS gives way to a year-named OS, I will provide yet another glimmer of light that may help free you from the dark side of embedded Internet. I'll use this newfound knowledge (for which I'm forever grateful) to continue our discussion about Agranat's EmWeb.

My non-year-named-laden, TCP/IP-addressed, Netscape-browser-loaded host PC is connected to a Net186 demo board that is serving an LED control page. So, let's move before my six months are up.

FORM AND STRUCTURE

Last time, I spouted off about EmWeb archives, HTML, and similar things that make EmWeb unique. This time, I spout off about EmWeb archives, HTML, and similar things with one minor difference—the addition of some 80186 hardware and HTML-based forms.

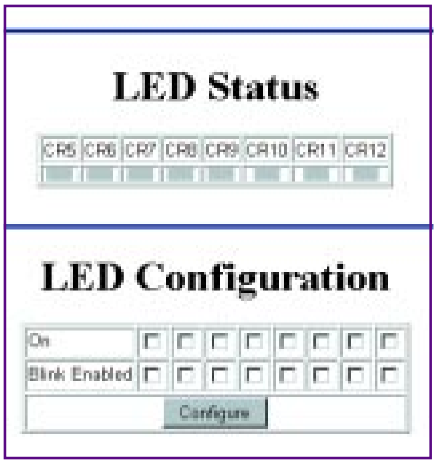


Photo 1—Thanks to the readers, here are the results of the HTML we will be examining.

Forms make the web interface. Everyone knows how to use controls on today's web forms. It's an intuitive process that's easy to use and understand.

The problem with web forms doesn't lie in the user community, but in the development community. Writing a good form is like writing a foolproof piece of code. Every input must be checked for validity, every error condition must be taken into consideration. You know the drill.

The designers of EmWeb feel your pain. One of its strengths is its ability to use the web form paradigm. The EmWeb form-u-la is to relieve the form-designer/content engineer from the duties of being a data cop.

Writing code to control the application with that input data is time better spent. Many an engineer will tell you that throwing money at a troublesome project may help, but often it doesn't.

Throwing code energy and logic at that same project is a more reliable way to get the job done. Agranat engineers

Photo 2—Here's a shot of the Net186 board for those of you that missed it the first time around.

must feel the same way. To ease the development and use of web forms and web form data, EmWeb throws C code and a couple major functions at the web form development task.

You already know that EmWeb uses regular HTML source laced with EmWeb tags to form part of the input module for the EmWeb compiler. And, since we're on the subject of forms, what tag could be better to begin a web form than `<FORM>`?

Adhering to a precise and logical form of programming, the EmWeb designers (like many others, including the TCP/IP inventors) placed their form data into a structure. As a matter of fact, its compiler creates a data structure every time a `<FORM>` tag is encountered in the HTML source file. But, this is no ordinary C structure. There are two substructures in each `FORM` data structure.

An example is better than a bunch of words, so consider the HTML statement `<FORM METHOD=POST EMWEB_NAME=CCINK>`. Let's take this statement apart from left to right.

Obviously, this is a `<FORM>` tag that would be parsed and acted upon by the EmWeb compiler. The `METHOD` attribute determines how the client's HTTP request will be formatted. Note that the prescribed method is `POST`. If the `METHOD` attribute is omitted, `GET` is the default method.

A `GET` instructs the browser to append query information to the request URL. This is how your everyday Internet search engine requests are constructed. Using `GET` enables the request to be cached for reuse without having to tell the user it's not live, it's Memorex.

On the other hand, `POST` tells the browser to include form submission data in the body of the HTTP request. Ella Fitzgerald is really singing here and the server is expected to respond. The response cannot be taped for later rebroadcasting or cached by one of those Intel guys in the funny suits.

EmWeb uses both methods, `POST` and `GET`, but `POST` is generally used in EmWeb applications. The reason behind this is that you are writing the form submission functions that ultimately return values for your routines to interpret and pass to the browser as visuals.

You are responsible for writing the function prototypes and the code. You're not out snipe hunting on the Internet. Things had better be where you put them (or left them) so you can go back and get them without fail.

That's what the EmWeb mechanism is looking for. As shown in Listing 1, its compiler concatenates `EwaForm_` with the `EMWEB_NAME` to create a name for the form data structure.

HARDENING OF THE FORM

At the beginning of this article I mentioned hardware and it's time to whip it out. You already know about the Net186 demo board because you read about it here, in *Circuit Cellar* 97. You remember, I was walking like an Egyptian in that one.

Anyway, if you missed it, the Net186 under the covers consists of a 40-MHz Am186ES microcontroller surrounded by an onboard Ethernet controller complex and a megabyte of memory divided equally as SRAM and flash memory.

The Am186ES integrates the functions of the CPU, nonmultiplexed address bus, three timers, a watchdog timer, chip selects, interrupt controller, two DMA controllers, PSRAM controller, asynchronous serial ports, programmable bus sizing, and programmable I/O (PIO) pins on one chip.

That's all good, but what really makes the demo board perfect for embedded applications is the onboard Am79C961A PCnet-ISA II Ethernet controller and the serial ports. As for cost, the Net186 won't put you in the bread line, and that's also a favorable factor for embedded projects.

The Am79C961A is a single-chip Ethernet controller with a built-in ISA bus and PHY layer (Manchester encoder/decoder and 10Base-T transceiver). This remarkable IC can also be configured for Ethernet full-duplex operation.

The Net186 also supports Magic Packet Technology. Magic Packet isn't a top hat and rabbit, but it is a neat box of tricks. It enables remote wakeup of a sleeping system on an Ethernet node.

The Net186 can be introduced to your special peripherals through a set of semi-PC/104 pins. It's amazing that all of this processing power is tied together with a single PAL. There's plenty of interest in the Net186 these days, and you can find this demo board in many other embedded application areas.

I'm a pushover for Ethernet, but the real thing that attracted me to the Net186 demo board was the LED array. Those eight side-by-side beacons really light up the bench at night!

Although I'm just kidding about the LED attraction, the LEDs will play a larger-than-normal part here because they'll help me describe how a real EmWeb application is put together. But you know I'm not kidding about Ethernet. Moving on....

I like LEDs and apparently someone at Agranat does, too. So, let's control the LED array on the Net186 board the EmWeb way. Photo 1 is our browser's view of the eight LEDs located on the Net186 board.

Photo 2 is for those of you that came in late. (Remember when your instructor used to use that line on you? Some professor is probably still using it on the younger of you out there today.) The idea here is to build a simple application to darken, lighten, or blink any or all of the LEDs in the Net186 LED array.

HTML is HTML in any other situation, but for our purposes, each line is important. Photo 3 shows the HTML for the LED status display. The CRX entries correspond to the LEDs on the Net186 assembly. Note here that each LED is off as denoted by the color of the `.gif` descriptions directly following the CRX entries.

Listing 1—In this skeleton,* `boolean led1` *represents a checkbox on the browser. The status bits that describe the state of the checkbox are kept in the* `uint8` *value.

```
typedef struct EwaForm_CCINK_s
{
  struct
  {
    boolean led1;     /* data values go here */
  } value;

  struct
  {
    uint8  led1;       /* status flags go here */
  } status;
}
EwaForm_CCINK;
```

Photo 4a builds the On checkbox table and Photo 4b does the same for the Blink checkbox.

You may be wondering about the <FORM> tag syntax at the bottom of Photo 3: <FORM METHOD=POST ACTION="/ledurl">. Everything has been covered left to right up until the word ACTION. Here's the scoop on that.

Normally, the ACTION attribute identifies the URL of the program to be invoked after the submission of a form. It usually redirects the submission to another web site via an absolute URL entry (http://something.com) or in our case to a program or local host file via a host-relative URL that begins with a slash.

The inclusion of the ACTION attribute enables multiple forms to be used in a single EmWeb document. This attribute is also used extensively in EmWeb applications to absolutely identify a form within a document and assure the form a unique place in the URL tree.

Although we're not privy to the content of /ledurl, we can deduce that it is located within the URL structure of the EmWeb Archive we will be using, and it contains code snippets and/or definitions. There is no form name because we're looking at the page served in return, not the original page that was served.

Wouldn't it be nice for the initial form to come up with some default values that are programmer specified and not junk characters that happen to inhabit that particular field's chunk of memory? This is where the first of the two functions that complement the form data structure come into play.

One of serve's tasks is to provide these default values, writing them into the EwaForm_EMBED_NAME data structure. Remember, EMBED_NAME is the name of the form that is joined with the EwaForm_text string.

Structures of data out there in memory land are nice to look at in source listings, but they need to be addressed and accessed to be useful. In the case of EmWeb, the corresponding form points to each data structure. Any default values outlined in the HTML source are used to initialize the EwaForm_EMBED_NAME structure prior to its release to the serve function. serve can use the defaults as if they were its own or override them.

Earlier, I mentioned a set of field-shadowing flags found within the data structure. I was referring to status flags, and you can see them in Listing 1. If a default value is changed or initialized by serve, the EW_FORM_INITIALIZED bit in the field's status flag must be set by the programmer to indicate to EmWeb that an initialization or change to that field's value took place. Other flags are set to indicate error conditions. Table 1 lays out all of the status flag options.

Once all of the decisions are made and the correct buttons or checkboxes are selected, the form's data must be submitted. Just like serve, the owning form points to the data structure that contains the field data entered and submitted by the user and passed to the submit function. This is where the field status bits earn their living.

When a user alters the specifics of a particular field (checking a box or entering text), the field's corresponding status flag has the EW_FORM_RETURNED bit set. In some cases, EmWeb provides some error checking of the syntax of input data.

If an error is detected, submit could be notified by the setting of the EW_FORM_PARSE_ERROR bit in the corresponding field's status flag. In our example, the submit declaration looks like <INPUT TYPE=SUBMIT VALUE="Configure">.

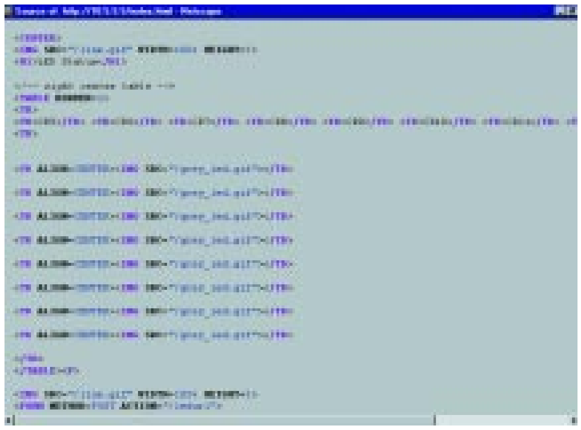


Photo 3—Here's how Photo 1 looks on the HTML side. Remember, this HTML is a response, not an original.

The VALUE attribute is actually the submit button's caption. This feature allows the submit button to be reused or relabeled dynamically, depending on what form is loaded and the message that needs to be conveyed.

OK. This is where the form hits the table. Selecting a checkbox alters the data field and as a result sets the EW_FORM_RETURNED bit in the associated data field's status flag.

For example, suppose the left-most checkbox was selected in the LED Configuration panel and Configure was clicked. The data (i.e., our check in the checkbox) would be submitted. In effect, by clicking on Configure in the browser, a request is entered for an HTML document and our checkbox data is submitted.

As a result of this request or submission, the EmWeb Server sends as much regular HTML content as it can until it encounters an EMWEB_STRING tag. This tag is associated with a piece of executable C code.

The EMWEB_STRING tag is not parsed. Instead, EmWeb executes code represented by the tag. In our example, the C code called by the submission checks the status flags for each data field or checkbox.

The benefit of browser checkboxes as they pertain to EmWeb is that instead of having to test various flags, the code only has to test the EW_FORM_RETURNED bit for that checkbox. If the EW_FORM_RETURNED flag is set, then the browser checkbox is checked.

In our case, the left-most checkbox is checked and it corresponds to LED CR5 in the LED

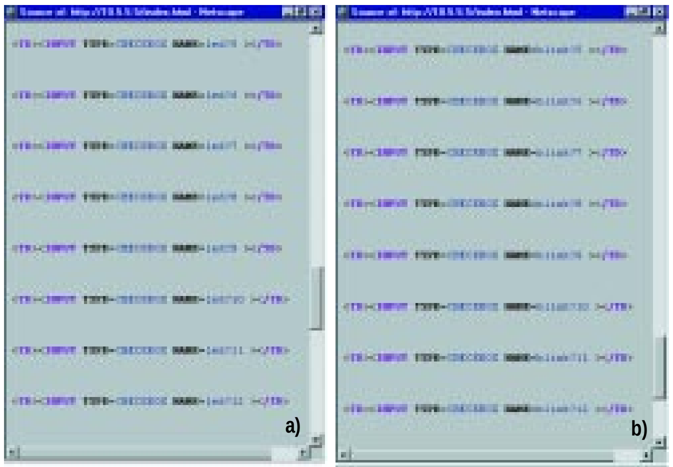


Photo 4a—None of the checkboxes are CHECKED here and thus none of the LEDs are On. The CHECKED attribute is returned with the served page to put a check in the browser boxes. b—Basically, this is the same logic and HTML that applies in (a). The names have been changed to protect the innocent.

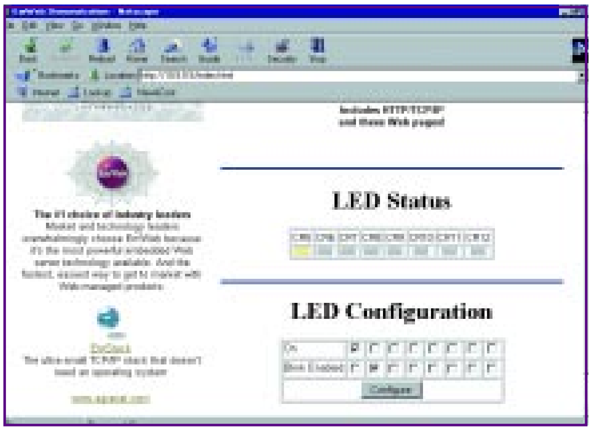


Photo 5—CR5 is blinking. What are the odds of me catching it off versus on? Note too, that this page was served from the Net186 and includes plenty of additional HTML stuff.

Status panel. The EW_FORM_RETURNED status bit is set for the data field representing LED CR5.

The program snippet in control here transfers control to the code that inserts the correct GIF (green for on, gray for off, green and gray toggle for blink) into an HTML page that will be served to the browser in response to the form's data submission. If any other boxes were checked, this process of GIF insertion would be done for all checked boxes.

Also, the CHECKED attribute is added to each checkbox HTML statement to indicate to the user that the box was indeed checked. The next step: perform the operation to set the Net186 LED, CR5, to the state selected via the browser. This is done via a code snippet embedded in the HTML source with the EmWeb tags.

Now, the HTML page that was built from the checkbox status routines is ready to be served to the browser. All of the values that should be returned are passed to the server where they're included in the document and served to the browser. Photo 5 plus a little Agranat HTML on the side is the result of the whole operation.

FORMING A PERSPECTIVE

We began with the need to control hardware via the Internet. The tools at hand were HTML, TCP/IP, HTTP, and the web. There was a single web-bound HTML document stranded without a ride. Enter the web-vehicle, EmWeb.

Suddenly, our everyday standard HTML document became a smarter everyday standard HTML document. EmWeb added the capability to include executable C code into that simple but smart everyday standard HTML document.

In an instant, the ghosts of scripting processes that lurked within our HTML remote control code disappeared. An archive containing all types of C-code-executing web content could be achieved by compiling our C-laden HTML source with the EmWeb compiler.

By the way, EmWeb also offers built-in TCP/IP and HTTP capabilities that can function without an operating system.

Constant	Value
EW_FORM_INITIALIZED	0x01
EW_FORM_DYNAMIC	0x02
EW_FORM_RETURNED	0x10
EW_FORM_PARSE_ERROR	0x20
EW_FORM_FILE_ERROR	0x20

Table 1—It's not a misprint. *EW_FORM_PARSE_ERROR* ***and*** *EW_FORM_FILE_ERROR* ***share the same flag bits.***

FORMING AN OPINION

I hope that this series on Internet control will help you get started in generating your own Internet appliance. In future columns, I'll continue to explore this increasingly popular area of embedded hardware and software.

I'll leave you with this thought: I was just reading a piece about children learning basic electronics, programming, and web concepts using Barney as the teaching vehicle. As far as I'm concerned, that proves it once and for all: It doesn't have to be complicated to be embedded. APC.EPC

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

SOURCES
EmWeb
Agranat Systems, Inc.
(978) 461-0888
Fax: (978) 462-1080
www.agranat.com

Net186 Eval kit
Advanced Micro Devices, Inc.
(818) 878-9988
Fax: (818) 878-9081
www.amd.com

FEATURE ARTICLE

John Iovine

Taking Orders

A Speech-Recognition Module

This versatile speech-recognition module has many uses, ranging from video game controls to home control for the disabled. Listen up as John explains how easy it is to train the module and implement it in an application.

The VoiceDirect module from Sensory is a great introduction into the world of speech recognition. The module, shown in Photo 1, is inexpensive and flexible.

VoiceDirect is stand-alone capable, ready to be embedded into your project. Or, it can be made to function as a slave under a host processor with enhanced speech-recognition capabilities.

In stand-alone mode, the module can recognize up to 15 words or phrases lasting up to 3.2 s each. Working as a slave under a host CPU, the module can recognize up to 60 words. Communication between the master CPU and the speech-recognition module takes place over a three-wire serial interface.

When a trained word is recognized, the module outputs a digital signal corresponding to the word recognized. The output line(s) associated with the word is brought high for 1 s. This signal may be used to control external devices with minimal external hardware.

The module is designed to be embedded into electrical switches, appliances, and consumer electronics. Before I delve into the features of this particular module, let me first define a few speech-recognition terms.

SPEAKER DEPENDENCE

Speech recognition is classified into two processing categories—speaker dependent and speaker independent. Speaker-dependent speech-recognition systems are trained by the person who will be using the system. These systems achieve a high command count and better than 99% accuracy for word recognition.

One drawback, however, is that the system responds accurately only to the individual who trained the system. But, an important advantage is that the circuit may be trained in any language.

Actually, language isn't even necessary. A series of grunts and whistles (as long as they can be repeated accurately) can be used in place of words. This is helpful to people who, through accident or illness, have lost the ability to verbalize words.

The VoiceDirect module is speaker-dependent, which is the most common approach employed in software for PCs. Sensory also offers other chips for use in speaker-independent modes.

A speaker-independent system is trained to respond to a word regard-

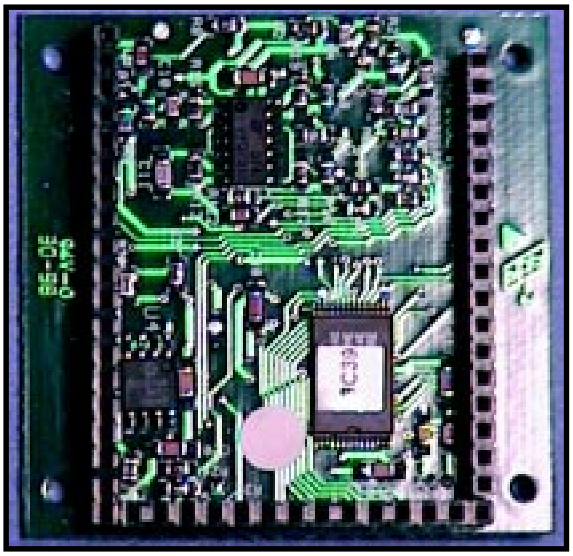


Photo 1—*This view of the VoiceDirect module circuit board shows the connection headers, which enable easy integration into your next project.*

less of the speaker. This system must respond accurately to a large variety of speech patterns, inflections, and enunciations of each command word.

The command-word count is typically much lower than the speaker-dependent systems, but high accuracy can be maintained when system demands are constrained by a limited number of commands. Industrial applications more often require speaker-independent voice recognition systems, such as the systems used by AT&T and other telephone companies.

RECOGNITION STYLE

Speech-recognition systems deal with another factor concerning the style of speech they can recognize. There are three distinct styles of speech recognition—isolated, connected, and continuous.

Isolated speech-recognition systems only handle words that are spoken separately (i.e., the user must pause between each word spoken). Ideally, the word is isolated by a moment of silence before and after it is spoken. This is the most common speech-recognition system available today.

Connected systems are a halfway point between isolated word and continuous speech recognition that permit users to speak multiple words. The VoiceDirect module recognizes verbalizations up to 3.2 s long. However, there shouldn't be a pause or period of silence longer than 0.5 s during the verbalization.

Continuous speech is the natural conversational speech we use everyday. It's extremely difficult for a recognizer to sift through speech because the words tend to merge together (a string like "Hi, how are you doing" sounds more like "Hiowryudoin"). Continuous speech-recognition systems are on the market and are under continual development.

VoiceDirect PROCESSOR

The heart of this module is the VoiceDirect speech recognition processor. The processor is available in a QFP-64 package for anyone who needs or wants to build a circuit from scratch.

This module has a lot of the preliminary work already provided, including an AGC audio amplifier, serial EEPROM, and clock. It also contains extensive socket headers (JP1, JP2, and JP3), which make it easy to connect an external circuit to the module. Table 1 shows you the pinout.

We can best explore the capabilities of the module by getting it up and running. The VoiceDirect speech-recognition kit contains the module (assembled), microphone, speaker, three microswitches, two 100-kΩ resistors, and a quick setup guide (see Photo 2).

The module's schematic is shown in Figure 1. The few external components that are supplied with the kit are all it takes to get the module functioning. The PCB measures 2″ × 2″ and has 0.1″ header sockets soldered to one side of the board, which makes it easy to connect to the circuit. To experiment with the module, I placed my external components on a solderless breadboard (see Figure 2).

To connect components on the breadboard to the socket headers, I used 22-AWG stranded wire. I mounted the microswitches, resistors, microphone, and LEDs on the breadboard.

Photo 2—*The kit includes the module, resistors, three push-button switches, microphone, speaker, and manual—all the external components you need to implement stand-alone recognition.*

One note here: the schematics in the manual detail the board shown from the top, but the header sockets are mounted on the bottom side of the PCB. This is in contrast to Figure 2, which shows the board with the header sockets on top. Be careful when comparing Figure 2 to the drawing in the manual.

The VoiceDirect module recognizes 15 words in stand-alone mode and has only eight outputs (connector JP2 pins 12–19). For simplicity, I'm only using eight outputs (eight words) so I don't need to add a decoding circuit. The 8-pin output for the 15-word recognition doesn't follow the standard binary numbering (see Table 2).

Name	Module	Pin	Description
	JP1	1–17	Unused
	JP2	9	Unused
	JP3	1–9	Unused
Preamp in	JP2	1	Microphone input connection
Mic Bias	JP2	2	Mic bias (electret microphone)
AGND	JP2	3,5	Analog ground—do not connect to digital ground because of noise
+5V	JP2	4	V_{CC}
PWM1	JP2	6	Pulse width modulator—Output 1, connects to 8–32-Ω speaker
PWM0	JP2	7	Pulse width modulator—Output 0, connects to 8–32-Ω speaker
DACOUT	JP2	8	Analog audio output—Provides better quality sound than PWM output, requires amplifier
RECOG	JP2	10	Recognition sensitivity selection and activates recognition
TRAIN	JP2	11	Training sensitivity selection and activates training
Out1–Out7	JP2	12–18	Stand-alone mode output lines 1–7
High/Out8	JP2	19	Stand-alone mode output line 8 / or high
ERROR	JP3	10	Stand-alone mode error signal
GND	JP3	11,12	Digital ground, CPU core (pins 1, 33)
Mode	JP3	13	Stand alone or slave
Reset	JP3	14	V_{CC}

Table 1—*Here you see the pinout for the module. The schematic is given in Figure 1.*

TRAINING THE MODULE

Training begins when the Train pin is pulled to ground for at least 100 ms. When you press the momentary contact switch marked Train, you are prompted to say the first word to be trained.

Speak the word or phrase you want the circuit to recognize into the microphone. It may be up to 3.2 s long but may not contain silences longer than 0.5 s. For example, "Circuit Cellar" is acceptable as long as the two words are not separated by a long pause.

Next, the module prompts you to repeat the word or phrase. Each time a word is entered, the module creates a template. The two templates for each target word are compared and, if similar enough, are averaged together and stored in memory. If the templates are too different, an error is generated, and the module asks you to repeat the word starting with the initial template.

Table 2—*This table shows the digital output for 15 words when the device is in stand-alone mode.*

Word 1 :	Output 1	Word 9 :	Output 8 and 1
Word 2 :	Output 2	Word 10 :	Output 8 and 2
Word 3 :	Output 3	Word 11 :	Output 8 and 3
Word 4 :	Output 4	Word 12 :	Output 8 and 4
Word 5 :	Output 5	Word 13 :	Output 8 and 5
Word 6 :	Output 6	Word 14 :	Output 8 and 6
Word 7 :	Output 7	Word 15 :	Output 8 and 7
Word 8 :	Output 8		

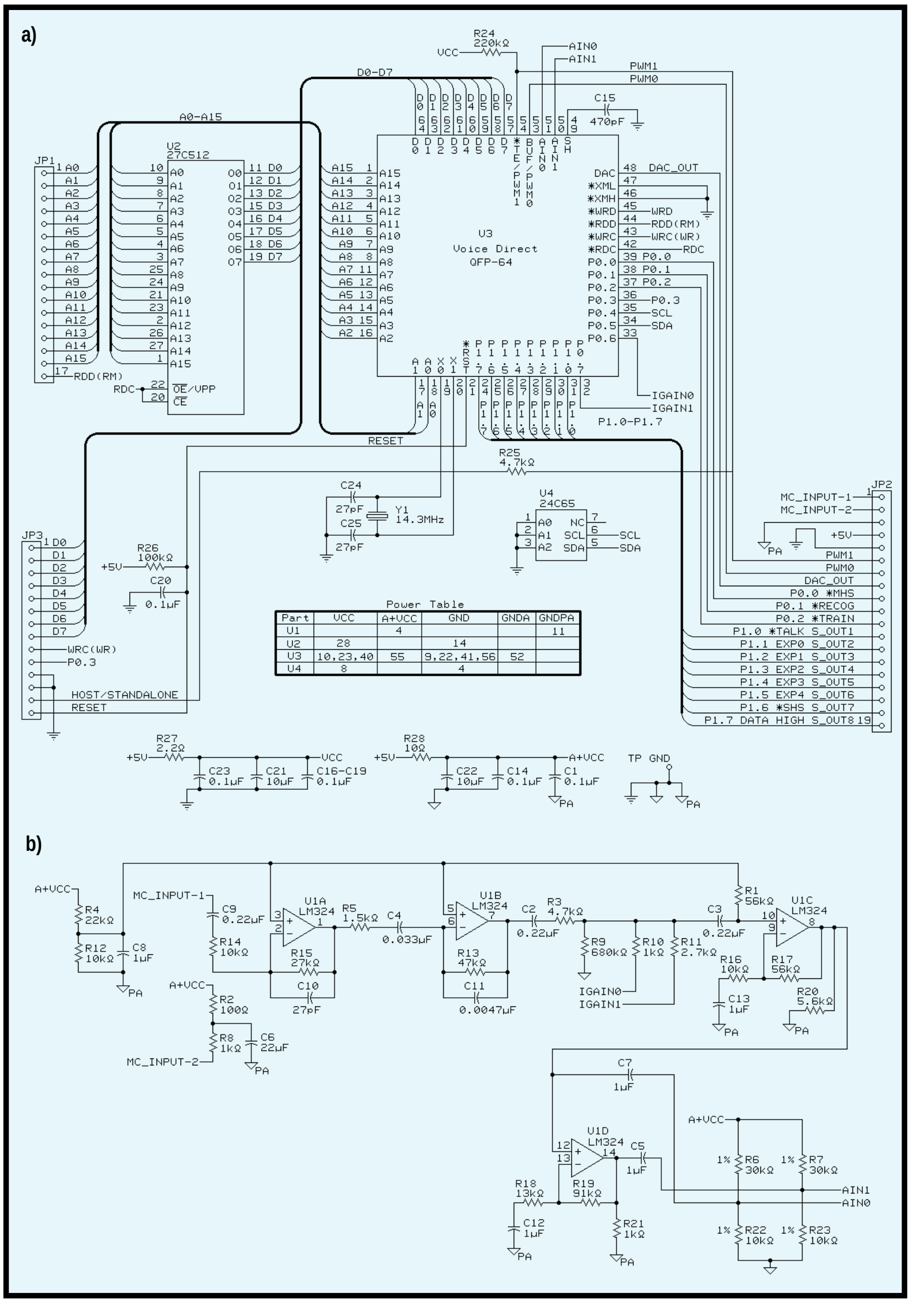
Before storing a word template, the new template is compared to the word templates already in memory. If the new template is too close to an existing template, the word is not accepted.

The VoiceDirect module has an automatic gain control over the audio amplifier to provide optimum signal strength. It also monitors the background noise and gives a warning if the noise is too high. A steady background noise (like a fan) has less impact on recognition than a fluctuating one (like a radio). Of course, best recognition occurs in low-noise environments.

Once a word is accepted, the module continues training by asking for more words. Training can be interrupted or stopped at any time by not speaking into the microphone at the prompt or by pressing either the Train or Recognize button.

Training is resumed by pressing the Train button. The module automatically starts training new words at the end of the previously trained words. For instance, if you trained six words and then stopped, when you resume training, the module automatically begins training at word seven.

Individual words and phrases cannot be erased or overwritten. But, the entire set of words can be deleted by simultaneously pressing the Train and Recognize buttons for at least 100 ms.



Power Table

Part	VCC	A+VCC	GND	GNDA	GNDPA
U1		4			11
U2	28		14		
U3	10,23,40	55	9,22,41,56	52	
U4	8		4		

Figure 1a—*As you can see from this schematic, the designers put everything on the board.* ***b***—*This section shows you the input amplifier with automatic gain control.*

RECOGNITION ERRORS

There are two common errors associated with speech recognition—rejection (failure to recognize a target word) and substitution (recognition of a non-target word or confusion between two target words).

When the module detects errors, it pulses the error pin high for 1 s. The LED connected to pin 10 on JP3 signals this condition. Errors also initiate a verbal response like "Spoke too soon," "Please talk louder," "Please talk

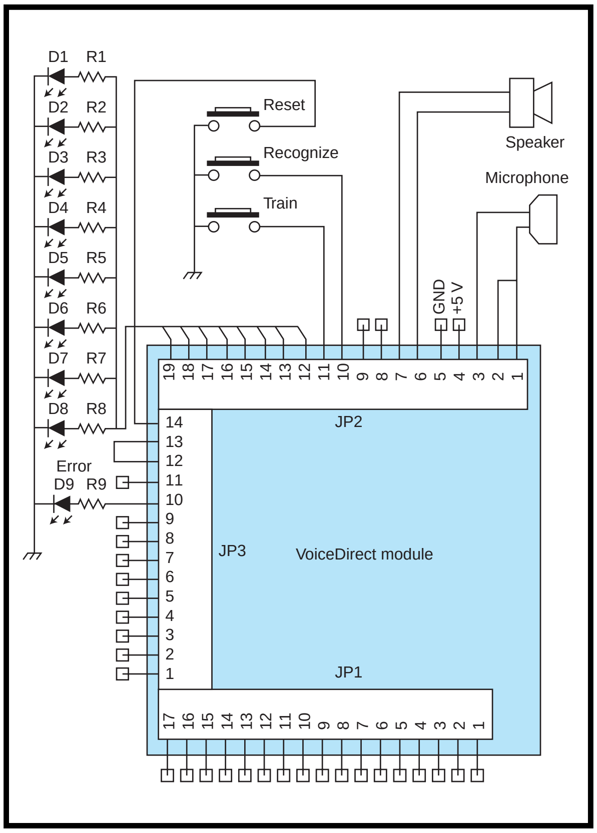


Figure 2—*This test circuit schematic demonstrates how simple it is to get the VoiceDirect module up and running.*

softer," and so on as the ERROR line is pulse high for 1 s.

The message "Word not recognized" is not handled as an error. If the module isn't trained on the word that initiated this message, it's not really an error.

The module has the ability to increase its selectivity. Figure 2 is configured for Relaxed Training and Relaxed Recognition. On power-up (or reset) the Train and Recognize pins control the selectivity.

If a 100-kΩ resistor is bridged across the Train switch, which essentially pulls the Train pin to ground with a 100-kΩ resistor, the module enters Strict Training mode. In this mode, the module rejects more similar-sounding words, resulting in better recognition of the words accepted.

Pulling the Recognize pin to ground with a 100-kΩ resistor places the module in Strict Recognition mode. The module recognizes fewer words and may reject trained words (fewer substitutions).

In the schematic, both the Train and Recognize pins are left floating (open circuit), which places the module in the Relaxed mode.

IMPROVING RECOGNITION

There are a number of ways to optimize recognition. Word selection is one primary technique—avoid homonyms such as red, bed, said and so on. In most cases, a synonym or approximate synonym can be used. For example, use "crimson" or "scarlet" in place of "red."

Another way to improve recognition is to match the equipment to the environment. The type of microphone you use to train should be the same type used for recognition. The distance from the microphone to the speaker's mouth should be approximately the same for training and recognition.

Keep in mind that your voice changes under stress or excitement. Imagine you're creating a voice-controlled joystick to fly your favorite military flight simulator. Your voice will sound quite different when you're sitting at your desk calmly programming your voice into the chip versus when you're engaged in a dogfight yelling, "Fire! Fire! Bank left!" You have to emulate the stress and excitement you feel while playing the game when you're programming the commands.

INTERFACING CIRCUIT

Depending on the application, you can design various interfacing circuits as well as a generic circuit. Because the output lines (OUT1–OUT8) only remain high for 1 s, some type of latching circuit is needed. For mine, I used one half of a 4013 dual D flip-flop.

The flip-flop, shown in Figure 3, turns the transistor on and off each time a command is given (e.g., if you connect the OUT1 [JP2–12] from the module to pin 3 of the flip-flop).

The first time the command associated with word one is given, the flip-flop turns on the TIP120 transistor that lights the LED. The LED remains lit until the command associated with word one is given a second time, turning off the transistor. This way, the same word can turn appliances on and off, and other commands given won't affect existing commands.

KEEP TALKING

The VoiceDirect module provides a good introduction to the world of speech recognition. It's easy to implement, easy to train, easy to use—what else is there to say?

John Iovine is the research director at Images Co. You may reach him at john@imagesco.com.

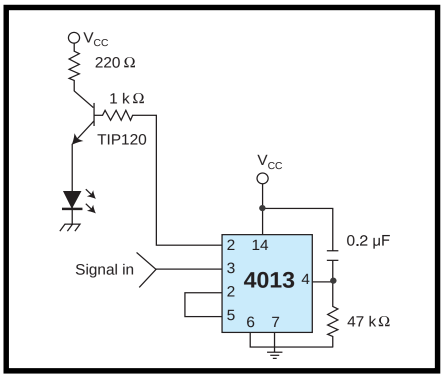


Figure 3—*The 4013 flip-flop latch helps control multiple appliances connected to the VoiceDirect module for speech recognition.*

SOURCES

VoiceDirect module
Sensory, Inc.
(408) 744-9000
Fax: (408) 744-1299
www.sensoryinc.com

All Electronics Corp.
(888) 826-5432
(818) 904-0524
Fax: (818) 781-2653
www.allcorp.com

Images Company
(718) 698-8305
Fax: (718) 982-6145
www.imagesco.com

Jameco Electronics
(650) 592-8097
Fax: (800) 237-6948
www.jameco.com

FEATURE ARTICLE

Gordon Dick

Induction Motors

Part 2: Working with Output

Put on your running shoes and take off with Gordon as he reduces noise, displays the frequency, and adds an analog isolation amp before finishing this VFD exercise he introduced last month. Talk about going the distance!

When I embarked on this project, I had no intention of taking it this far. I did the work described in Part 1 because I was interested and because I felt a small need to redeem myself for not finishing the original project so many years ago.

When I got ready to do the work I discuss this month, I thought it would go quickly because I had the parts on hand and had already checked out the application circuits and knew they were simple. How many times have I started out this way and then discovered how wrong I was because I hadn't anticipated this or that? This project was no exception.

The project described here is a prime example of how electronic engineering has changed with the inclusion of intelligence in so many parts. The circuitry added to the VFD is rather simple—on the exterior anyway. The time needed to do the wiring is minimal. The major effort is producing code to configure and customize the VFD board.

THE ISOLATED 15-BIT ADC

The scheme used to measure motor current involves the HCPL-7860 isolated modulator and the HCPL-7870 digital interface IC. This method is appealing because of the high resolution and the three-wire serial interface to the controller.

Of course, the set provides isolation which, in this application, is essential rather than nice to have. Because I hadn't played with anything like this before, I worked with this chipset first.

Before getting into the details, it's appropriate to spend some time examining what's contained in this chipset and talking about the operation. The technical data document from HP explains things well (as does Figure 1), so here's a small part of it (with some minor changes and omissions):

The Isolated Modulator converts a low bandwidth input signal into a high-speed one-bit data stream by means of a sigma-delta oversampling modulator. This modulation provides for high noise margins.

The modulator data and on-chip sampling clock are encoded and transmitted across the isolation boundary where they are recovered and decoded into separate high-speed clock and data channels. The Digital Interface IC converts the single-bit data stream from the Isolated Modulator into 15-bit output words and provides a serial output interface that is compatible with SPI, QSPI, and Microwire allowing direct connection to a microcontroller.

Now that you know a little about how this chipset works, let's build something that uses it. The circuit board for the basic VFD of Part 1 has no space for expansion.

I had extra space until I decided to include an EPROM and run the 'HC11 in expanded mode. Now I'm glad the EPROM is there, but more on that later.

I considered building a new board with space for the additional circuitry, but not for long. The thought of redoing all that wirewrapping and the troubleshooting wasn't pleasant. It turned out to be practical to mount a daughterboard on top of the basic VFD board. Signals could be connected via a short ribbon cable and the header on the VFD board that was used for initial testing would be useful again.

After playing puzzle with the items needed on the daughterboard, a reasonable layout emerged and parts were mounted. A hole was necessary to provide access to the speed-control pot.

Wiring the circuit shown in Figure 2 didn't take long. Two small changes from HP's application circuit are: the elimination of the current-sampling resistor because it was already present in the IRPT1059C, and the addition of pullups on the three-wire serial link.

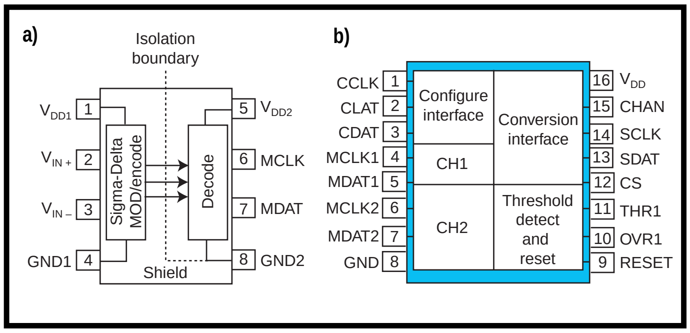


Figure 1—*These functional diagrams of the HCPL-7860 (**a**) and the HCPL-7870 (**b**) show a chipset with serious applications flexibility. Many of the features of the HCPL-7870 were not used here.*

Now it's time to create some code and do some testing. I'm using my trusty 'HC11 prototyping board. The daughterboard is connected to it via ribbon cable and we're ready to go. I'm trying to slow-step my way through the timing diagram shown in Figure 3.

Yes, the SDAT line goes low when CS goes low, but after that it looks like data is being transmitted. This is puzzling! The SDAT line should wait for clock pulses to be applied to the SCLK before it starts sending data.

I pondered over this for a good while and almost called applications support, thinking that they must have left something significant out of the datasheet. Finally, I decided that the device was designed not to wait for me to do things slowly and would go ahead on its own. They could've said so in the datasheet.

With this incorrect assumption, I went ahead and constructed code to configure the SPI and initiate data transfers. As I soon found out, this was a serious mistake.

To test this code, I made the input to the 7860 zero to see if it converted correctly. The datasheet indicates that zero should convert to 4000h and when the most-significant bit (which is used as a start bit) is taken into account, the reading should be C000h (the converter code is 0000h for negative full scale, 4000h for zero, and 3FFFh for positive full scale).

When the code was run, I got data from the 7870 but it was erratic. With some imagination I can see the reading C0C0h coming up a lot among all the other readings. The first part looks correct but why another C0h? To make things worse, when I connect the scope to look at what's happening, conversions stop.

When I added tags to the code to indicate where it might be hanging, the added instructions change the way the system behaves. Now it runs most of the time the scope is connected.

Now that I can use the scope, some things are starting to make sense. First off, I can see that the SPI is not getting two data transfers during the time CS is low. My code doesn't display a value until two SPI transfers have taken place, which currently requires two CS low times. No wonder the data is appearing as C0C0h.

After studying my code, I found a couple of serious logic errors which, when corrected, allow two SPI transfers to happen when they are supposed to. Things have improved now to the point where the reading from the converter is now C0*xx*h where *x* indicates that it could be anything.

During a discussion with a colleague about this project, my earlier assumption about the 7870 not waiting was challenged. My colleague maintained that after a conversion is done, nothing should happen on the SDAT line until clock pulses are provided on the SCLK line.

To test this idea, the SCLK line was removed from the 'HC11 and shorted to ground at the 7870. When CS was asserted, the SDAT line went low then high and stayed high, just as the timing diagram shows. Then it all started to make sense.

The reason the 7870 appeared not to wait for a clock must have been because of noise. Now that I'm thinking about it, yes, the signal lines are noisy.

The wires between the 7870 and the 'HC11 are quite long. The ribbon cable between the 'HC11 board and the daughterboard is 17″ and there's nearly another 9″ of wire from the header to the 'HC11. I figured it was likely that these long wires were allowing noise to be picked up.

I moved the daughterboard as close to the 'HC11 as possible and hardwired to it. What an improvement! The data being read is now C00xh. Quite a lesson to be learned here.

I was fairly confident that I had the ADC working properly, but there was still more noise in the reading than I wanted. It was more than just the lower four bits being uncertain—more like the lower six bits—but I decided that would be sufficient for the moment. When the daughterboard was mounted to the VFD board, it would be even closer to the 'HC11 and that should reduce the noise further.

As a final test, I connected a variable voltage to the converter input and it responded as expected. Moving on....

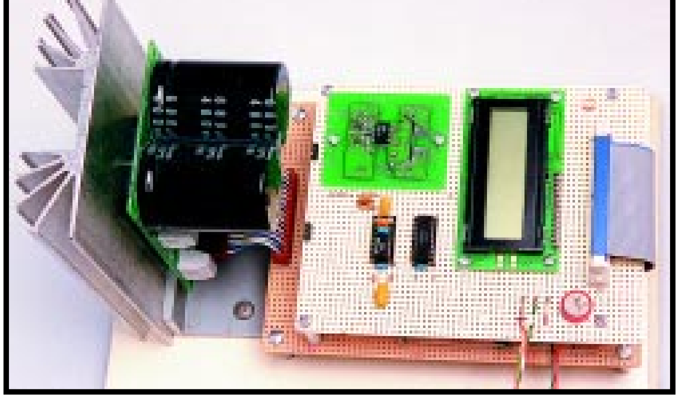


Photo 1—*The daughterboard piggybacks on the VFD board. Motor current and voltage signals enter via a pair of three-pin headers on the left, and micro signals connect via the ribbon cable on the right.*

COUNTS TO CURRENT

The LCD had been wired to the daughterboard some time ago (see Figure 4) and I'd been using it to test the 7870, but the display was raw. I was just sending hex data to it for testing; nothing pretty. Time to change that.

Displaying the voltage and frequency would be easy. The frequency was already available from the 'HC11 ADC and would require little manipulation. The voltage portion wasn't done yet but when it was, the voltage number would also be available from the 'HC11 ADC system and it too wouldn't need much massaging. But it appeared to be a different story with the current.

Whenever I'm confronted with a new problem, I almost never get to solve it the easy way. It seems that I have to do it the hard way before an easier solution surfaces. Such was the case with the current display.

My calculation showed that this converter produces a conversion number of 2250 counts/A. If I wished to display the current in amps I needed to do a division of 2250. But a plain old integer divide (idiv) wouldn't do. I needed the fractional part.

I had never used fractional divide (fdiv) before, but as I read about it, I decided it was just what I needed. Doing an idiv leaves a remainder, and following that with an fdiv gives me a suitable fractional part which, after converting to decimal, completes the almost floating-point division. I ran a little test code just to be safe.

The only down side to my method was that I didn't have a routine to convert fractional hex numbers to decimal. Fortunately, the routine I used to convert hex integers to decimal could be easily modified to do fractions. It wasn't long before I had a reasonably attractive display showing amps with a fraction and a place for frequency and current to be displayed next.

REDUCING THE NOISE

I'm starting to think that I have this project complete so maybe I should try to get rid of that noise in the displayed value of the current. Earlier I set up a circular buffer and was averaging it in an attempt to reduce the noise. At only four elements, it was too small. There wasn't much of an improvement with or without it. Looks like more values needed to be averaged.

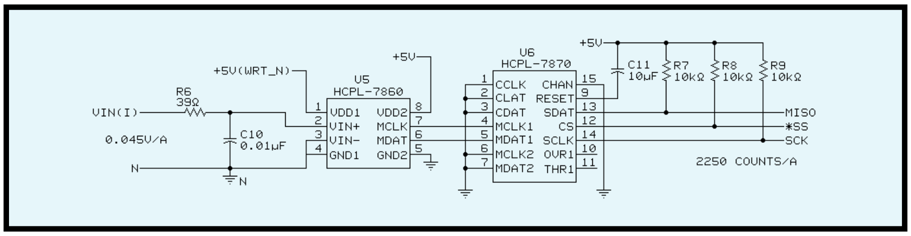


Figure 2—*This is HP's suggested circuit with only minor changes. Note that the ground symbol with the N adjacent to it is the common for the IRPT1059C, which needs to be isolated from the other grounds.*

This averaging process gets cumbersome rather quickly. Values produced by the 7870 in this application are always positive, so I did away with that sign bit and had numbers ranging from 0000h to 3FFFh after masking off the start bit.

When averaging only four numbers (which can never be larger than 3FFFh), the sum prior to division still fits in 16 bits. Two left shifts and the average is obtained. But, averaging more numbers introduces sums larger than 16 bits and shifts that span more than 16 bits if the division remains a power of 2.

For this second version of averaging, I could use a 16-element circular buffer that required 24-bit addition, which I already had a routine for. The only new code needed was a 24-bit left shift.

After some testing, I was confident my new averaging code was working properly. So, I inserted it into the routine that reads the 7870 and displays the result.

Would you believe that the program to read the 7870 crashes now? It crashes every time it's run and in a different way each time. It was almost like a subroutine wasn't returning properly. Every subroutine was checked for a missing `rts`, but they were all in place. I had no success until I begin watching the data being written into the circular buffer.

Data was getting written beyond the reserved buffer space and over the top of executable code. This explained why the crash was different each time. Although it took some time to find, fixing the problem was quick. The loop that gathers the readings was testing for 0x15 values being read in rather than just plain 15 values.

The good news is that the code is working once more. The bad news is that there isn't much improvement in the displayed noise. Looks like the displayed current needs to be truncated to two digits after the decimal. There is enough noise that keeping four digits after the decimal doesn't make sense.

It wasn't until some time later, during another discussion with a colleague, that the subject of converting this count value to current came up. I explained what I'd done, thinking I'd been fairly clever in how I handled the problem.

He suggested a multiplication of 1000 followed by a division by 2250 and then an artificial shift of the decimal point three places to the left. At first I thought this method was far simpler, and suddenly I didn't feel so clever. Later, however, I realized it wasn't that much simpler. I'll use it next time.

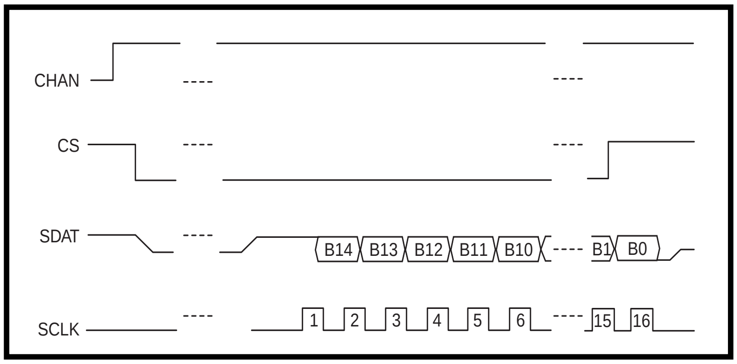


Figure 3—*The sequence of events on the SPI bus is illustrated with these waveforms. Observe that only 15 bits of data are obtained. The most significant bit is always high and will be discarded later.*

ANALOG ISOLATION AMPLIFIER

The last of the new circuitry is the analog isolation amplifier. The IRPT1059C provides an output signal proportional to the inverter bus voltage that will be the input for the analog isolation amplifier.

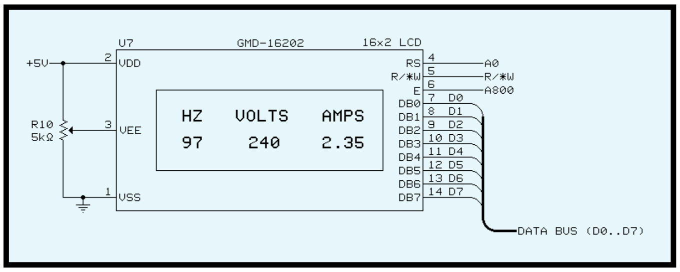


Figure 4—*In LCD wiring is straightforward. A two-line device allows for a pleasing display of frequency, voltage, and current.*

Because HP was kind enough to send me an eval board, I saved some time here (the eval board uses a HCPL-7820 isolation amplifier, which has some similarity to the HCPL-7860; a sigma-delta ADC is optically coupled to a DAC).

A single 5-V supply could have been used with the 7820 and its support circuitry. But, there are some differences between that circuit and the eval board, and because the eval board uses mostly surface-mount components, I was reluctant to attempt modifications. This meant that ±15 V had to be wired into the daughterboard.

Certainly if you're building a production prototype, the extra supplies can be avoided. The circuit I used is shown in Figure 5.

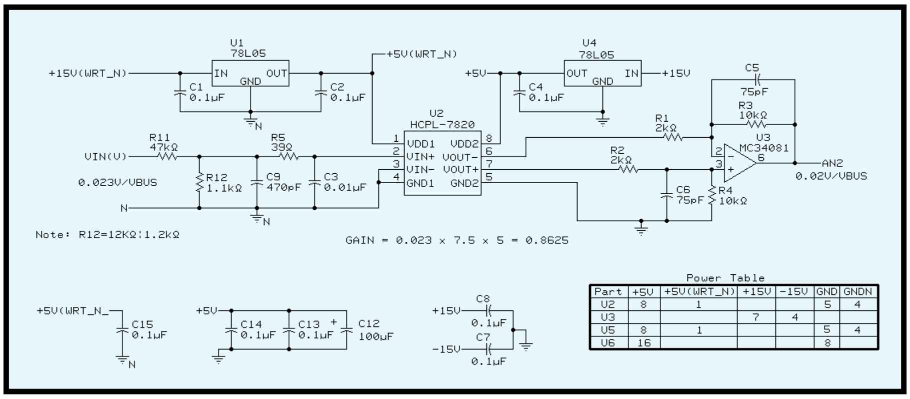
There's another aspect of this circuit that deserves some discussion. The gain of the eval board by itself is too high for this application. The 7820 is designed for a full-scale input of 200 mV and provides a typical gain of 8 (7.5 for my device) and the diff amp U3 provides an additional gain of 5.

The signal from the IRPT1059C can be as high as 5.52 V if the bus is at 240 V. Scaling this voltage down to 200 mV is not sufficient, because 200 mV × 7.5 × 5 = 7.5 V, which is too large for the ADC input of the 'HC11. Hence the input to the 7820 had to be further attenuated to produce a signal with a suitable range for the 'HC11 shown in Figure 5.

Again, this is something that would be avoided on a production prototype. It would be better to allow the 7820 to use more of its available range and reduce the gain of U3.

Earlier I had cut some excess off the eval board and mounted it to the daughterboard without wiring it. Now I wired it and added the necessary attenuator. After a few tests I decided it was working as it should. Just a little more code and I'd be finished with it.

Although the remaining code didn't have to do anything new in principle, it turned out to be somewhat clumsy to use routines designed for words on a byte to convert it to decimal, then to ASCII, and display it. I wrote a few routines here just to improve that, and I also improved the look of the display by eliminating leading zeros.



Power Table

Part	+5V	+5V(WRT_N)	+15V	−15V	GND	GNDN
U2	8	1			5	4
U3			7	4		
U5	8	1			5	4
U6	16				8	

Figure 5—*Here you see HP's eval board schematic with a minor addition. A front-end attenuator (R11 and R12) was added because the signal from the IRPT1059C was too large.*

DISPLAYING THE FREQUENCY

The SA828 is a write-only device (i.e., it can't be read to see what frequency it's running at). To display the frequency of the motor voltage, I simply needed to display the command frequency, which is convenient because the same routines that display the voltage can display the frequency.

At this point, the project is electrically complete, as shown in Photo 1. The code still needs tweaking though. The code was created as two separate modules—one for the VFD portion and one for the display portion.

The two modules now need to be integrated. Fortunately, there are only two conflicts to resolve between the modules (or so I think). The first is the way the 'HC11 ADC is initialized.

Instead of converting a single channel continuously, the integrated module converts a four-channel group continuously. That way, a conversion is always available and just needs to be read from the correct result register.

The other conflict is that the original VFD code used Port D signals PD4 and PD5 to reset the IRPT1059C and the SA828, respectively. Because the SPI is being used, PD4 and PD5 are now SCK and *SS. Luckily, the timer system is not being used here, so other I/O pins (OC2 and OC3) are available.

Other issues (particular to my development environment) had to be resolved before I got the integrated code to work. With these changes made, the system runs quite happily out of RAM on my development board.

Now it needs to be made ROMable. Because of what I'd already been through (see Part 1), it shouldn't take long to get the integrated code EPROMed and running. Once again, it wasn't quite that simple. This code uses RAM variables (in the 256 bytes internal to the 'HC11) that, in a few cases, had to be initialized. Neglecting to do that the first time produced a scrambled LCD.

Many times I looked back on the decision to include an EPROM as a fortunate one. The code for the first part fits into the EPROM at 282 bytes. But, the code for the second part needs another 1064 bytes. It's unlikely that it can be streamlined to fit into 512 bytes.

MORE IMPROVEMENTS, ANYONE?

One possible improvement is that the display portion would perform better if it were done on a PCB. The layout suggestions offered by HP in the app note could be followed, and the signals produced by the 7820 and 7860 would contain less noise.

Or, I could add another serial link between the 7870 and the 'HC11. At first glance, using this second serial port wouldn't be as simple as the SPI link. In my case, the 'HC11's SCI is used to communicate with the PC so another serial link would need to be created in software.

The third possibility is to leave the serial link in place so the 7870's conversion mode could be changed (there are five modes) to improve the SNR hence the number of useful A/D bits.

By using the serial link, the input threshold could be programmed and an interrupt produced when some level of sampled current is reached. This signal would appear at THR1 on U6. Threshold detection time is also software controlled.

Again, using the serial link, the overload level could be programmed and another interrupt produced, likely to indicate some fault condition. This signal would appear at OVR1 on U6. Or, I could use the serial link to make the 7870 execute an internal offset calibration routine.

Lastly, the pretrigger mode can be changed by using the serial link. This one's a bit complicated so I'll let the HP literature explain that option. ■

Thanks to Bernd Kohler, a colleague and friend, for his helpful discussions regarding some aspects of this project.

Gordon Dick is an instructor in electronic technology at the Northern Alberta Institute of Technology, Edmonton, Alberta, Canada. He occasionally consults in the area of intelligent motion control and is an avid hunting retriever trainer. You may reach him at gordond@nait.ab.ca.

SOFTWARE

Source code for this article may be downloaded via the *Circuit Cellar* web site.

REFERENCES

Hewlett-Packard, *Isolated 15-bit A/D Converter*, Technical data, 1996.

Hewlett-Packard, *High CMR Analog Isolation Amplifiers*, Technical data, 1996.

Hewlett-Packard, *Evaluation Board for HCPL-7820/HCPL-7840*, Technical information, 1995.

Hewlett-Packard, *Designing with Hewlett-Packard Isolation Amplifiers*, App note 1078, 1995.

SOURCE

HCPL-7820, **-7860**, **-7870**
Hewlett Packard
US Electronic Components Div.
(408) 654-8675
Fax: (408) 654-8575
www.hp.com

MICRO SERIES

Monte Dalrymple

Rolling Your Own Microprocessor

The Design and Debug Process

Part 1 of 2

With all the options available in the world of microprocessors, you might think it would be easy to find a micro to meet any need. Monte begs to differ. A custom processor designed with Verilog turned out to be the best solution.

What do you do when your microprocessor vendor isn't providing the features you need and you have many man-years invested in your software? These days, it's not unrealistic to think about designing your own processor.

Rabbit Semiconductor had already decided to do this when they asked me to help them with the design of a new processor. In this series, I discuss the steps I took and the features I incorporated into the project. Part 1 goes through the design and debug process, and next month I'll talk about the actual features of the chip.

USING VERILOG HDL

I don't mean to imply that designing a microprocessor from scratch is a trivial task, but the tools available today certainly make it easier than it was 15 years ago when I did my first processor. Back then, I drew every transistor, by hand, on D-size sheets of vellum. Today, I sit at my PC and type in a text file that describes the design using a hardware description language (HDL).

Two standard HDLs are available: VHDL and Verilog. I won't start a religious argument by saying that one is better than the other, but I prefer Verilog. It's a lot like C, whereas VHDL is more like C++. I use Verilog in this article, but everything I say here also applies to designing with VHDL.

Listing 1 shows an example of Verilog code that describes a byte-wide register. It has a clock, a reset input, a load enable signal, and input and output buses. Listing 2 shows an 8-bit counter. Describing a circuit is that easy.

You can either place these modules in your design like subroutines, or you can just place the four lines that describe its operation (the four lines starting with `always @`) as inline code. Of course, you have to give each register a different name with each placement.

Verilog also has all the logical, arithmetic, and shift operators to enable you to describe what you want done without worrying about the details of how to connect the gates. The Verilog description of the design is then fed into a logic synthesis tool, which automatically translates the Verilog into the necessary logic gates which are targeted to the technology specified by the designer.

This simplicity is another advantage of designing with Verilog. If you've done it right, you can easily retarget your design to a different technology. Later in the article, you'll see how I took advantage of this.

SPECIFICATION

The most important step in any design is to decide what you're going to design. You're probably thinking, "Duh...," but a poor job at this stage will lead to delays or even failure.

Like every other project, there were some specific requirements. For example, the processor had to be compatible with the Z80 instruction set (remember the man-years invested in developing the software). It also had to execute instructions faster than the Z180. That meant a two-clock-cycle basic instruction execution time.

Because it was for embedded applications, it needed a full suite of peripherals and a glueless interface to memory chips. The man-years of software had also exposed several areas for improvement in the instruction set, which would give significant performance gains, so those had to be included.

Listing 1—*This code is all it takes to do a byte-wide register in Verilog HDL.*

```
module reg_byte (clk, resetb, ld_byte, data_in, data_out);

  input        clk, ld_byte, resetb;
  input  [7:0] data_in;
  output [7:0] data_out;
  reg    [7:0] data_out;
  always @ (posedge clk or negedge resetb) begin
    if      (!resetb) data_out <= 8'h00;
    else if (ld_byte) data_out <= data_in;
    end

endmodule
```

All of this was specified, in detail, before a single line of Verilog code was written. For example, I had to work out the number of clocks for every instruction and the memory access operation for every clock. This information was then refined and expanded to include the internal data movement timing.

Once this step was complete, I did the Verilog coding. In many ways this is the easiest step of the process, especially if the specification has been done properly. At the end of the coding phase, the project was roughly half done.

Yes, half done, and now the fun begins. You have to check everything, preferably by two people working independently. I checked it all using a Verilog simulator, and my client checked everything using a breadboard.

SIMULATION

Another advantage of Verilog is that you can model a whole system and simulate it using any of a number of low-cost simulators. To debug and verify the processor, I placed it (still in Verilog format) into a test bench that was also written in Verilog.

The test bench contains a model of a memory to hold program code and data, but this model also checks all of the memory control signals out of the microprocessor on every read. If the processor tries to read from an undefined memory location, it gets undefined data, which shows up immediately in the simulator. This memory is read-only.

A separate memory that holds the expected data is used for writes from the processor. If the processor writes the wrong data to a memory location, or data to a wrong location, this memory model raises an error flag. This memory is also read-only, because it just looks at the memory write operations.

This arrangement means that I have to manage the memory carefully, so that each write is to a different location. By doing this, the test bench automatically checks for proper operation for both reads and writes.

Somehow I have to place the appropriate data in these memories. This step requires assembly-language code that contains every opcode, every special boundary condition between instructions, flag results, and so on.

The write memory must contain the expected data, which is usually dumped after every instruction. The task of creating, assembling, and debugging this piece of code is in fact larger than the task of designing the processor.

The one advantage here is that I can see every node inside the processor, so if it does something unexpected, it's easy to track down. Usually, I trace a set of about 200 nodes when I'm debugging (things like the PC, the machine state register, ALU control signals, and all the processor pin signals), so I have a pretty good idea of where to look when something goes wrong.

When a problem arises, it's easy to add signals in the area of the problem to the trace list. Then it's a matter of rerunning the simulation, figuring out what went wrong, and fixing the problem in the Verilog source code. This process usually takes a couple iterations.

You might be tempted to just trace everything all the time. Don't do it. Simulation time increases significantly with the number of nodes traced.

I think that 100–200 critical signals is a reasonable compromise. Half the time, the problem is in the assembly-language software of the test pattern anyway.

The other thing to consider is the length of the simulation. The full test pattern for the CPU portion of our processor takes a little over 50k clock cycles. Rather than doing it all at once, I divided the test into a number of individual patterns, each concentrating on one class of instructions.

Once the CPU is debugged, it's time to start on the peripherals. This processor has a full complement of parallel ports, serial ports, timers, counters, and so on. But because these peripherals aren't programmable except by the CPU, they require more assembly code to debug them.

The one thing that complicates debugging the peripherals is the fact that they respond to more external stimuli than the CPU does. That means a lot more Verilog code in the test bench. And the stimuli need to somehow be synchronized with the code that the CPU is running.

I accomplish this synchronization by having the test bench recognize an

Listing 2—*Making an 8-bit counter is a simple modification.*

```
module cnt_byte (clk, resetb, inc_byte, data_out);

  input        clk, inc_byte, resetb;
  output [7:0] data_out;
  reg    [7:0] data_out;
  always @ (posedge clk or negedge resetb) begin
    if      (!resetb) data_out <= 8'h00;
    else if (inc_byte) data_out <= data_out + 1'b1;
    end

endmodule
```

I/O write to a particular address as a sync signal. Then a separate Verilog `drive` file can keep track of program execution and provide stimuli and sample outputs at known times.

Because everything is relative to this sync signal, I don't have to count clock cycles from the beginning of the pattern—only from the nearest sync signal. Counting clock cycles is still the most tedious part of debugging though, because everything has to be correct down to the clock cycle. Such precision is necessary because these programs will later become the test vectors to test the final chips.

When the test patterns were complete, the design was ready for release to the semiconductor fab. However, because this was my client's first ASIC, we wanted independent verification that it was going to work. Hence the breadboard.

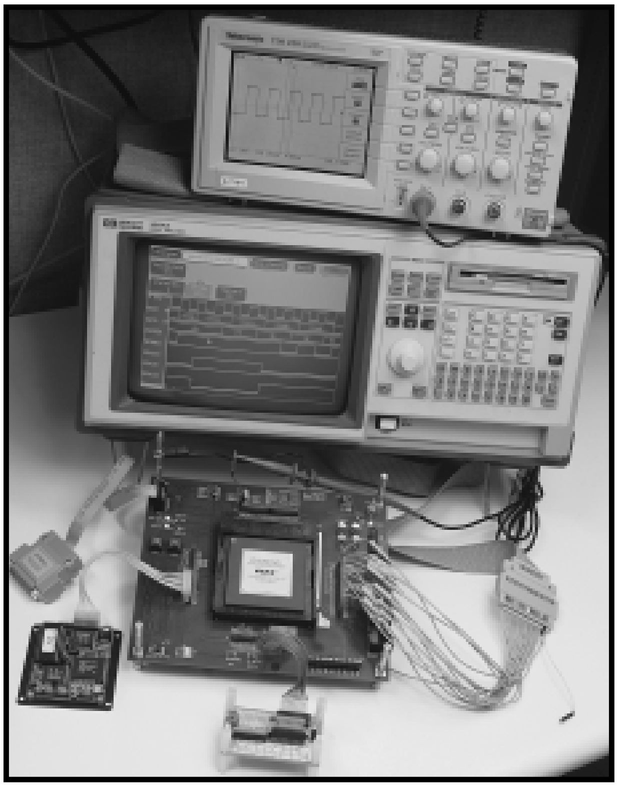


Photo 1—*Here you see my FPGA breadboard with the debugging equipment connected.*

BREADBOARDING

The breadboard's design and fabrication started at the same time as the Verilog coding. The breadboard is a full system based on our processor, but it uses a large FPGA in place of the finished processor (see Photo 1).

Designing in Verilog enabled us to use an FPGA quite easily. The only difference between the Verilog code for the released version of the processor and the FPGA version is the connection of the clock drivers in the FPGA, which is required only because our FPGA has dedicated clock driver pins. Any change required only a couple hours to recompile and reroute the design.

The full design of the processor requires slightly less than 25k gates, but it took a 130k-gate FPGA to hold the design. A large portion of a CPU is combinatorial in nature and the logic cells of an FPGA are balanced between combinatorial and sequential. Keep this factor in mind when you use an FPGA.

Debugging this system was a little more work than debugging a normal embedded system because there was always a question of whether or not the processor was working correctly. So we didn't even power up the breadboard until I had most of the CPU working in simulation. Even then, every time something came up on the breadboard, I had to go back to the simulation to make sure the circumstance had been tested.

The breadboard debugging first concentrated on the peripherals, using them just as they would be used in finished systems. Again, this required a lot of time with assembly language and a logic analyzer. Even though I was concurrently doing the same thing in simulation, the two tasks were kept separate to assure independent checking.

The breadboard permits more exhaustive testing because it runs closer to real time. The FPGA clock is limited to 8 MHz, but it's still many orders of magnitude faster than the simulation. My simulations run at the equivalent of about 8 Hz, even on a fast PC.

We never needed to take advantage of it, but an FPGA breadboard enables you to bring internal signals out of the FPGA. These chips have over 400 pins, but our design only has 88 signal pins, which leaves a lot of pins available for probing the inside of the design.

SOFTWARE DEBUGGING

As well, the software engineers used breadboards to port the development software and C compiler over to the processor. They realized that a minor change to a couple of instructions would help with some of the housekeeping necessary when dealing with the MMU.

Verilog enabled me to make these changes with the addition of just two lines of code. Of course, verifying the change, which dealt with sampling interrupts, took much longer.

An exhaustive instruction set test was assembled once the bulk of the development software and C compiler was completed. It checked every instruction with every flag combination and nearly every data value. Such a test wasn't practical in simulation, and it did find a problem with two instructions that were setting a flag when they were not supposed to.

Having the FPGA allowed most of the library routines, interrupt service routines, and communications handlers to be written and debugged before the processor went to the fab.

WRAPPING IT UP

Designing with Verilog and a synthesis tool is the only way to do a project of this scope. Next month, I'll detail exactly what our processor does.

Monte J. Dalrymple has been designing integrated circuits for more than 20 years. He currently has his own company which develops intellectual property. You may reach him at monted@systemyde.com.

REFERENCES

M. Keating and P. Bricaud, *Reuse Methodology Manual*, Kluwer, Dordrecht, 1998.

E. Sternheim, R. Singh, and Y. Trivedi, *Digital Design With Verilog HDL*, Automata Publishing Co., Cupertino, CA, 1991.

Verilog information, www.ieee.org, www.ovi.org, news:comp.lang.verilog

Get SmartMedia

FROM THE BENCH

Jeff Bachiochi

Part 1: What's It All About?

Pull a chair up to the bench as Jeff discusses the makeup of SmartMedia—one of the newest forms of nonvolatile storage. Who uses it? What makes it so different? With Jeff, we'll see the whole picture.

Documenting family life has been a popular pastime for decades. Still photography progressed through 8-mm home movies and camcorders, and now the circle is complete. The digital still camera has developed to the point where it can begin to compete with film.

But high-resolution picture files easily top a megabyte in size. I don't know about you, but the nonvolatile memory I'm familiar with is expensive and bulky. What happened?

Except for some Sony digital cameras that use 3½″ floppy disks for file storage, the format of choice is SmartMedia. Even if you've seen SmartMedia, you probably haven't given it much thought (and that's how it should be).

Although I balk at taking a screwdriver to my Toshiba PDR-M1, I'm curious about this new media—or rather, about its format. Lots of distributors are selling it, but there doesn't seem to be much information available.

SSFDC FORUM

The Solid State Floppy Disk Card Forum is one organization promoting SmartMedia. SmartMedia consists of a removable NAND flash-memory card that claims to be the lightest, thinnest, and cheapest of its kind in the world.

Weighing in at 2 g (less than 0.1 oz), SmartMedia is about the size of a matchbook but only 0.76 mm (less than 0.03″) thick. Compared to a PCMCIA memory card of the same capacity, SmartMedia costs about a third less.

Probably the most curious thing about SmartMedia is the contact configuration (see Photo 1). The 22 contacts are arranged in two rows of 11, embedded into the top surface of the device.

When you see that power and ground use up four contacts, you start wondering how any kind of meaningful storage can be accomplished via the remaining ones. They are shared by the I/O and control signals.

Treating the physical interface as an I/O device simplifies and standardizes the interface, but does make it a bit more difficult to use. Connections remain identical for any capacity of SmartMedia card.

Although SmartMedia's largest application is presently with digital cameras, that's only the tip of the iceberg. Others include PDAs, electronic musical instruments, voice recorders, and portable terminals. Practically any equipment that uses removable memory is a good candidate for SmartMedia.

FACE TO FACE

Let's take a closer look at the interface of these bite-sized storage devices (see Figure 1). The NAND flash array is set up in columns of 528 bytes, each column called a "page." Notice that this is a wee bit larger than an even 512 bytes. The 16 extra bytes in each page are not meant for data storage.

A page is the smallest portion of memory that can be programmed (i.e., written to). To allow this to happen smoothly, a page buffer holds data to be written to or read from any flash page.

Stack up 32 pages and you have a block (512 × 32 + [16 × 32] = 16K + 512 bytes). A block is the smallest portion of memory that can be erased.

SmartMedia is not considered byte-writable. To change 1 byte (or all 528 bytes) in a page requires that a whole block be read or erased and that any pages of data be replaced with the updated data.

The smallest SmartMedia devices are 2 MB, which gives 128 blocks. The 64-MB SmartMedia now in development requires 4096 blocks of flash memory.

GETTING AT 'EM

Having a data path of only 8 bits requires that the large addresses possible with SmartMedia be broken up into byte-sized pieces, transferred, and reassembled internally. For the SmartMedia to know what's coming through the pipe, the user must use the format of a small, predetermined command set.

The SmartMedia commands are read (flash memory), write (to data buffers), program (data buffers to flash memory), erase (flash memory), reset, and status (see Figure 2). All commands require one or two bytes of command data. Most require address data to follow.

Command bytes are written with the command latch enable (CLE) high and the address latch enable (ALE) low, whereas address bytes require the opposite states. The SmartMedia circuitry grabs the data presented on the 8-bit I/O bus on the rising edge of the write enable (WE).

When the first byte sent is command 90H with a following address byte of 00H, the SmartMedia retrieves two bytes from a special ID area. These bytes can be read from the SmartMedia device by lowering CLE and ALE and bringing the read enable (RE) line low once for each byte. The data is stable 35 ns after the falling edge of RE.

The first byte is a manufacturer's code and the second byte is a device code. These bytes are important because they indicate the type and capacity of the storage medium.

SmartMedia devices with up to 32 MB require three address bytes. Larger devices (up to 8 GB) need four. Read the ID to determine how many address bytes are required.

In addition to the ID, SmartMedia has two read commands. The read command transfers a complete page from the flash memory to the data registers. The first address byte (A0–A7) is a pointer to within the data register. The second, third, and potentially fourth address bytes indicate which page to transfer (A9–A*xx*).

You may have noticed that A8 is mysteriously absent from the address bytes. But, recall when I described the page size as 528 bytes; A8 isn't needed unless you wish to point to a starting address within the upper 256 bytes.

The `read1` command holds A8. Use the command 00H for pointing to within the lower 256 bytes and 01H to point to within the upper 256 bytes.

Prior to getting the page data from the SmartMedia, you must wait 10 µs. This can be accomplished by watching the ready/busy output.

After receiving the last address byte, the ready/busy output goes low until the page is transferred from flash memory. When the ready/busy output goes high, you may read bytes bringing the RE input low for each byte you wish to read.

Access to the data registers begins at the address A0–A7 + A8 in the command byte and may continue up to the end of the page. You can continue reading up to the end of the block, but remember to wait for each page to be transferred before continuing to read.

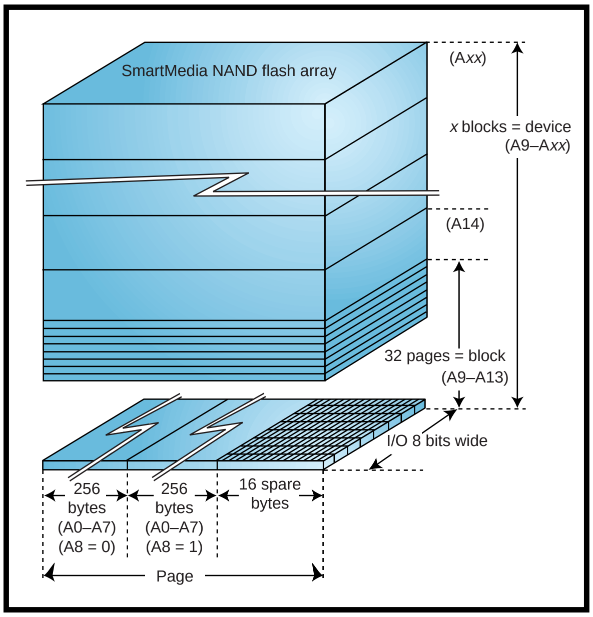


Figure 1—*Unlike conventional memory, each page of data has its own spare area associated with it.*

528 – 512 = 16

Here's where we learn about those 16 extra bytes in each page. Because manufacturing these large-scale NAND flash memory arrays is difficult, there's the possibility of having a defective cell in one or more pages. A cell may also deteriorate during its life of over a quarter million programming cycles.

Photo 1—*The SmartMedia shown here on top of the 3½″ and 5¼″ diskettes look like toys, but each one holds many times the capacity of their predecessors.*

The extra 16 cells are a way to keep track of bad pages, like bad sectors on magnetic media. When the SmartMedia is tested after manufacturing, if a bad page is found, a 00H is written to the sixth extra byte of the last 16 bytes of the page (the other 15 bytes are erased to FFH). Be advised that when a block is erased, so are the bad page identifiers, so you must keep track externally and rewrite the indicators after erasing.

The status register can be read by issuing a command 70H and reading back the status byte (after 60 µs). Three of the eight bits are significant. Bit 0 indicates a program/erase success with 0 and a failure with 1. An erase failure indicates that all bits within the block could not be erased to an FFH state.

A program failure means at least one byte's bit within the page could not be programmed to a 0. Now, you can update that page's invalid page indicator so it won't be used in the future.

Bit 6 is a software indication of the hardware ready/busy output—0 when busy and 1 when ready. Bit 7 indicates the protect status (0 is protected and 1 means not protected).

The protection status is an input pin to the SmartMedia. A low on this pin protects the flash memory from being erased or programmed. You may place an adhesive conductive dot on the SmartMedia to indicate that the device is read-only. This conductive

CLE	ALE	*CE	*WE	*RE	*WP	Mode	
H	L	L	⊔↑	H	X	Read mode	Command input
L	H	L	⊔↑	H	X		Address input 3 bytes (or 4)
H	L	L	⊔↑	H	H	Write mode	Command input
L	H	L	⊔↑	H	H		Address input 3 bytes (or 4)
L	L	L	⊔↑	H	H	Data input	
L	L	L	H	↓⊔	X	Sequential read and data output	
L	L	L	H	H	X	During read (busy)	
X	X	X	X	X	H	During program (busy)	
X	X	X	X	X	H	During erase (busy)	
X	H/L	X	X	X	L	Write protect	
X	X	H	X	X	H/L	Stand-by	

Figure 2—*This chart shows how the control lines are used for each of the SmartMedia commands.*

dot is read by a pair of contacts on the SmartMedia socket and must be interpreted by your hardware.

FILL'N IT UP

I don't know about you, but to me, erasing something means getting rid of everything. But with memory devices, erasing is filling each memory cell with 1's. This is more like charging than like erasing memory.

And when it comes to programming, you need only worry about cells that have to be 0. Merely poke a hole in the bottom of each appropriate cell and let the unwanted 1 drain out.

A block (16K) is the smallest portion that can be erased at once, so only addresses A14–A*xx* are needed. Following the command byte 60H, only two address bytes (or three for 32+ MB SmartMedia) need to be passed.

But erasing will not commence automatically unless the command D0H follows the previous command and address bytes. This prevents unintentional erases. You can read the status to determine when the erasure has completed and whether or not any erasure errors were discovered.

Writing to the SmartMedia is similar to erasing a block, but it's handled on a page basis. Like reading, you may start anywhere within the page. The pointer is A0–A7 (the first address byte), with A8 being the least significant bit of the command (here, 80H or 81H).

Following the rest of the address, A9–A*xx* (which selects the page and block), there is an "are you sure" command necessary as with erasing. This command byte, 10H, begins the page programming process. And, as with erasing, you can read the status to determine when the programming has completed and whether or not any program errors were discovered.

PHYSICAL VS. LOGIC FORMAT

The physical format is all you need to write data to and read data from SmartMedia. All SmartMedia is preformatted; all pages are erased to FFH, with any invalid pages marked at location 517 (6 bytes into the spare 16-byte array) with a 00H. To help standardize SmartMedia and ensure compatibility, the devices should be logically formatted in a familiar file structure.

Although SmartMedia is not byte writeable and the I/O structure makes it slow for random access, it's ideal for applications requiring lots of storage. In Part 2, we'll look more at the SSFDC Forum and I'll demonstrate how you can interface to SmartMedia in a project of your own. ■

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.*

REFERENCE

SSFDC Forum, www.ssfdc.or.jp/english/index.htm

SOURCES

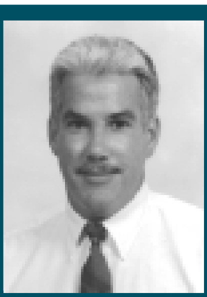
SmartMedia
Toshiba
(408) 737-9844
Fax: (408) 737-9905
www.toshiba.com

Samsung Semiconductor, Inc.
(408) 544-4000
Fax: (408) 544-4907
www.usa.samsungsemi.com

SILICON UPDATE

Tom Cantrell

PIC Up the Pace



It's a close race on the fast track of the MCU industry, and Microchip's latest entry is looking to the inside on the leader. Join Tom as he shows us how this new PIC can tune up our applications.

What a rags to riches story, eh? Who would have thought that a lowly '70s-vintage controller would be such a hit? Resurrection of the PIC has rocketed Microchip from obscurity to number two in the 8-bit MCU biz, breathing down longtime leader Motorola's neck.

Microchip continues to expand and freshen the PIC lineup—a wise move since the MCUs account for 80% of their sales. Competition is intense, and no one can afford to rest on their laurels.

Say hello to the new PIC18C*xxx* shown in Photo 1. If the original PIC16C5*xx* I wrote about back in '92 was the mini-pickup of MCUs, then the '18C*xxx* can be considered a Land Rover. With lots more luxo-features and options, it's still able to get the job done when the going gets tough.

MICRO MAKEOVER

Of course, an MCU with upscale pretensions needs a strong motor, which is why the '18C*xxx* offers significant improvement in performance, instruction set, memory capacity, and organization (see Figure 1).

Speed has always been a relative advantage for the PIC over most other 8-bit MCUs. PICs might not be able to do everything, but what they can do they do rather quickly. Back in '92, the 5 MIPS (four clocks for most instructions at 20 MHz) on tap were a notable advantage over the 1–2 MIPS of other popular chips.

However, the years since have seen the PIC speed advantage slowly but surely eroding compared to both historic competitors (i.e., turbo-51's from Dallas and Philips) and new contenders (Atmel AVR, Scenix SX, etc.).

To stay near the front of the pack, the '18C*xxx* boosts performance to 10 MIPS by jacking the clock rate to 40 MHz. Along with the traditional clock options (crystal/resonator, oscillator, external RC), the '18C*xxx* incorporates a clock that quadruples the PLL.

It works with a commodity 4–10-MHz crystal (i.e., 16–40-MHz CPU clock) while reducing EMI concerns and leaving headroom for future speedups. Another new addition is an extra on-chip oscillator that can be selected as an alternate CPU and peripheral clock under software control.

Memory addressing and organization were, to put it politely, never the PIC's strong suit. Designed back when memory cost more than peanuts, the original PIC architecture didn't foresee, nor deal particularly well with, the integration of more than a few kilobytes of memory.

The '18C*xxx* takes advantage of the new instruction set to expand program space to 2 MB (i.e., one million 16-bit instructions) and data space to 4 KB by adding a two-word instruction format for handling extended branch offsets and RAM addresses. And to make life easier for both ASM and C programmers, three dedicated index registers with autoincrement and -decrement support a software stack.

Although a bit ironic, it's a fact that RISCs are becoming more CISCy all the time—and the '18C*xxx*, with more than twice the instructions of the original PIC (75 vs. 33), is no exception. Some may chafe at the fact that the '18C*xxx* is only source-code-compatible with earlier parts, but the need to execute old binaries is mainly a problem for PCs, not MCUs. And it's not as though the extra instructions, including stalwarts such as bit manipulation and single-cycle multiply, aren't welcome.

For example, handling table lookups on the original PIC posed a problem faced by Harvard architectures with physically separate program (ROM/OTP/flash memory) and data (RAM) memories. It's easy enough to store a table or string constant in nonvolatile program memory—just include it in the .HEX file when you burn the chip. But getting at it is another story, because the program memory can't be accessed as data.

Some clever PIC guru of yore figured out a jump-table hack that exploits a variant of the RETURN instruction (RETLW) that specifies a byte literal to be loaded into the W register. To find the *n*th byte in a table, you do a computed CALL to TABLE+*n* where a RETLW instruction with the desired value resides. Very clever, but also very ugly (and wasteful, consuming a 16-bit RETLW opcode for each 8 bits of data).

While purists may argue that the new '18C*xxx* table-lookup instructions (using a 21-bit pointer register for individual-byte access to the entire 2-MB code space as shown in Figure 2) are counter to RISC theology, long-suffering PIC programmers will no doubt breathe a sigh of relief.

INTERRUPT DRIVEN

I still remember being bemused by the fact that the original PIC didn't have an interrupt input. By contrast, the '18C*xxx* includes a total of 17 interrupt sources, including on-chip peripherals and three dedicated pins (INT0–2). There's a Port B change interrupt that detects activity on four pins of Port B (RB7–4), which is useful for keypads and buttons.

A two-level (high and low) priority scheme supports nested interrupts (i.e., a high-priority request can interrupt a low-priority service routine).

Interrupt response is fast and predictable at 3–4 instruction cycles (i.e., 300–400 ns at 40 MHz). The response time is the same whether single- or double-word/cycle instructions are underway when the interrupt occurs.

However, vectoring to the interrupt routine is only part of the story. Besides stacking the return PC, other critical registers (i.e., the W and STATUS registers) typically must be saved. To expedite matters, the '18C*xxx* incorporates a one-level-deep fast interrupt stack for the PC, W, and STATUS. If not needed for interrupts, the stack is exploited by new fast CALL and RETURN instructions for speedy subroutine entry and exit.

The longest journey starts with reset and the '18C*xxx* includes everything you'd find in an external supervisor chip, including low-voltage detection and power-on reset, oscillator startup, and watchdog timers (see Figure 3).

The watchdog timer has its own RC oscillator (18-ms timeout, with 1:1, 1:2...1:128 postscaler), so it can run even when the chip is in Sleep mode, which shuts off both the primary and alternate oscillators. Reset and an interrupt or watchdog timeout will wake up a sleeping '18C*xxx*.

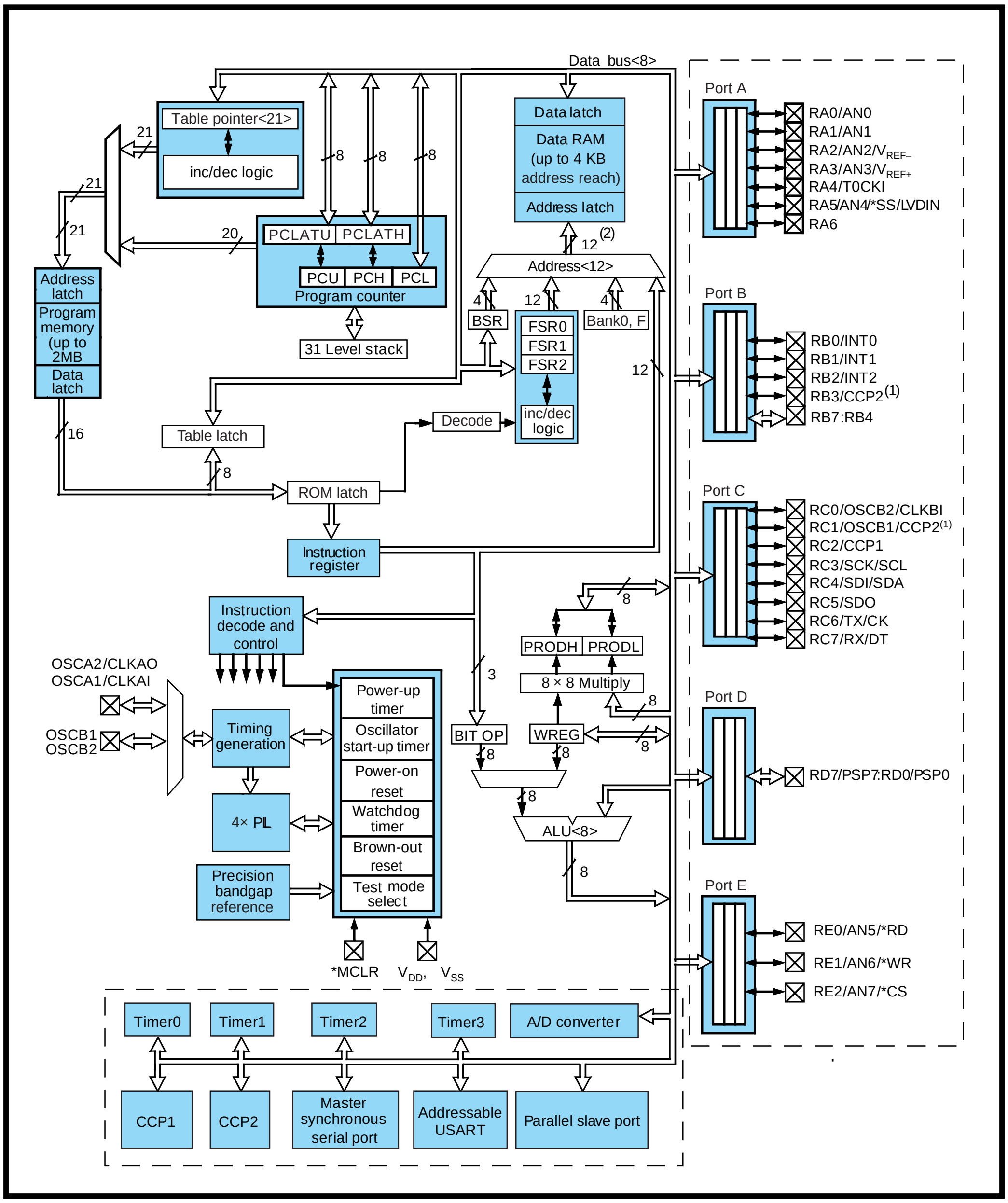


Figure 1—*Initially, versions of the '18Cxxx are available with 16/0.5 KB or 32/1.5 KB OTP/RAM, in regular (4.2–5.5 V) or low-voltage (2.5–5.5 V) versions and in 28- or 40+ pin DIP and surface-mount packages (the 40+ pin parts include Ports D and E).*

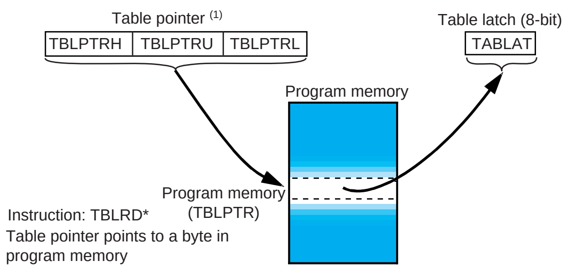


Figure 2—*The `TBLRD` instruction, using a 21-bit pointer register, makes it easy to read data (such as strings and lookup tables) stored in program memory. There's also a `TBLWT` instruction that, for future versions of the chip that support it (i.e., flash memory) could modify the program memory under software control.*

PERIPHERAL INTERFACE CONTROLLER

That's what General Instrument, a company subsequently scattered to the wind, had in mind when they came up with the PIC way back when. It was designed to handle I/O for their high-hopes-at-the-time minicomputer chips.

Because the PIC's original purpose was to interface peripherals, it didn't need any of its own. By contrast, the '18C*xxx* comes with a full quiver of I/O, including timer/counters with capture, compare, and PWM; serial ports; A/D inputs; and a parallel slave interface.

The 28-pin '18C*xxx*s include three 8-bit ports (A, B, and C) while 40- and 44-pin chips add ports D and E (the parallel slave interface). To accommodate all the I/O functions, practically every pin has a multiplexed special function. Note that '18C*xxx* devices are pin-compatible with their like-packaged '17C*xxx* counterparts.

TIMER ZONE

The '18C*xxx* has a total of four timer/counters with a variety of features and operating modes.

Timer 0 is a holdover for compatibility with the good old days. It's a simple 8-bit timer/counter with 8-bit prescaler. In timer mode, it increments every instruction cycle (i.e., 100 ns at 40 MHz). In counter mode, it increments every rising or falling edge (selectable) on a pin (RA4).

Compatibility with the past includes warts and all. Be advised, as legions of PIC programmers have learned the hard way, that writing the Timer 0 data register automatically clears the prescaler, which by the way can neither be read or written directly.

I suggest that Timer 0 is best assigned to simple set-it-and-forget-it-tasks. Save the fancy stuff for the other timers that are of much more recent vintage.

Table 1—*Running at 40 MHz, Timer 2 and the CCP (compare/capture/PWM) module work together to maximize PWM frequency and resolution.*

PWM Frequency (kHz)	**2.44**	**9.74**	**19.53**	**39.06**	**78.12**	**208.3**
Timer prescaler (1, 4, 16)	16	4	1	1	1	1
PR2 Value	0xFF	0xFF	0xFF	0x3F	0x1F	0x17
Maximum resolution (bits)	10	10	10	8	7	5.5

Timers 1 and 3 are 16-bit units (all 16 bits readable and writable) that, like Timer 0, count instruction cycles or pin edges. In addition, both can optionally select the alternate oscillator (OSCB, typically 32 kHz) as a clock source.

Timer 2 is an 8-bit unit with programmable pre- and postscalers, clocked each instruction cycle. It includes a comparator that clears the timer whenever the count reaches a programmed value (i.e., essentially an auto-reload).

Timers 1, 2, and 3 work in conjunction with a pair of capture/compare/PWM (CCP) modules, each of which includes a 16-bit register, programmable prescaler (1:1, 1:4, and 1:16), and dedicated pin.

In capture mode, the value of the timer (1 or 3) data register is written to the CCP register whenever the capture condition (i.e., every rising edge, every fourth falling edge, etc., on the CCP input pin) is detected.

In compare mode, the CCP pin is an output that's updated (set high, low, toggled, or unchanged) whenever the timer (1 or 3) data register matches the compare value programmed into the CCP register.

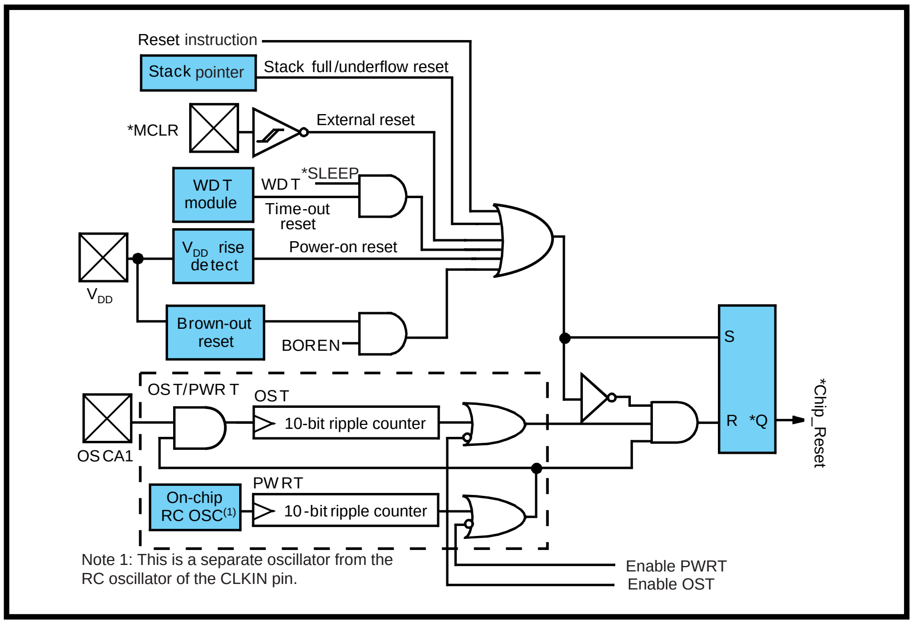


Figure 3—*The '18Cxxx reset circuit eliminates the need for external discrete components or supervisor ICs.*

Meanwhile, Timer 2 and the CCP work in concert for PWM. In this case, one CCP register defines the PWM period and another the duty cycle. Table 1 shows examples of PWM frequency/resolution tradeoffs. Note that the PWM registers are double-buffered, so a change in the period or duty cycle won't take effect until the current cycle completes, which avoids the possibility of glitches during the transition.

SERIAL SOLUTIONS

Although PIC programmers have honed bit-banging to a fine art, there's no doubt that many applications have better use for their MCU than babysitting a serial port in software.

To that end, the '18C*xxx* has both a clocked serial port (known as a Master Synchronous Serial Port [MSSP]) and an async port (called an Addressable USART [AUSART]).

The clocked serial port is versatile, able to handle both of the popular standard protocols, I^{2}C and SPI. In particular, it fully supports the more complicated protocol features in hardware.

For instance, I^{2}C multimaster mode requires that each master perform arbitration to make sure another master isn't trying to butt in. Each master must confirm that the bus in fact reflects what they are trying to send (i.e., a master has to listen at the same time it is talking). Without the specific support that the '18C*xxx* provides (i.e., hardware-collision detection), I^{2}C multimaster mode would be quite a challenge using software alone.

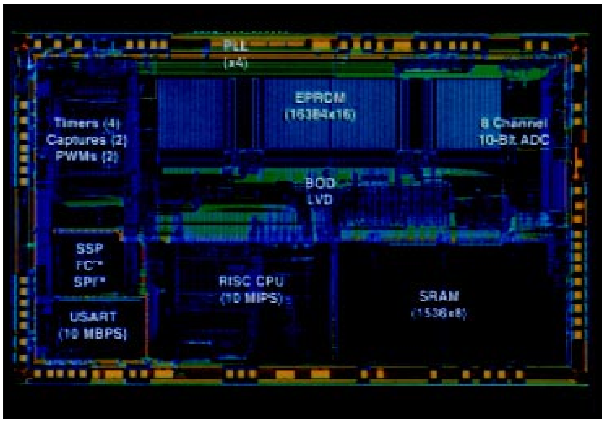


Photo 1—*The new PIC18Cxxx family surrounds an enhanced CPU core with plenty of memory and I/O.*

The "A" (Addressable) in AUSART refers to the fact that the async port supports the ninth-data-bit mode, suitable for implementing a multidrop bus (e.g., RS-485). Typically, the ninth data bit is used to distinguish between address and data packets such that devices on the bus can ignore messages that aren't addressed to them.

Though the AUSART has overrun and framing error detection, parity is left as an exercise for the programmer.

The serial ports, both sync and async, incorporate their own data-rate generators, so one of the more valuable timers (0–3) doesn't have to be devoted to the cause (note: TMR2 can optionally serve as the MSSP clock source).

At higher MCU clock rates, practically any required transfer rate from fast (10 Mbps) to slow (300 bps) is possible, though top speed and breadth of selection is reduced at lower clock rates.

For in-box host connections, the larger (40+) pin '18C*xxx*s include Ports D and E which, in addition to straight parallel I/O, can be configured as a bytewide parallel slave interface (see Figure 4). Port D is the 8-bit data while three lines of Port E function as *RD, *WR, and *CS handshake connections with the host.

BEYOND 1S AND 0S

To top it off, the '18C*xxx* includes a multichannel (five inputs for 28-pin parts, eight for 40+ pin parts) multiplexed input 10-bit ADC. Either the positive or negative rail can be specified as a reference or an external reference can be used as well.

The source of the A/D conversion clock is programmable as a multiple of the MCU clock. The conversion clock should be programmed such that the minimum allowed conversion time (1.6 µs per bit) isn't violated.

In addition to running off the main MCU clock, the ADC also incorporates its own RC oscillator that can be called on to clock the conversion. Although the conversion timing isn't especially accurate (typically 4–6 µs per bit, but can vary between 2 and 9 µs per bit), the independent RC oscillator uniquely allows A/D conversion to proceed while the MCU is sleeping (i.e., MCU clock shut off).

That's great for high-speed designs, especially those running on batteries. Lots of power can be saved by sleeping during conversions instead of executing hundreds of instructions that may

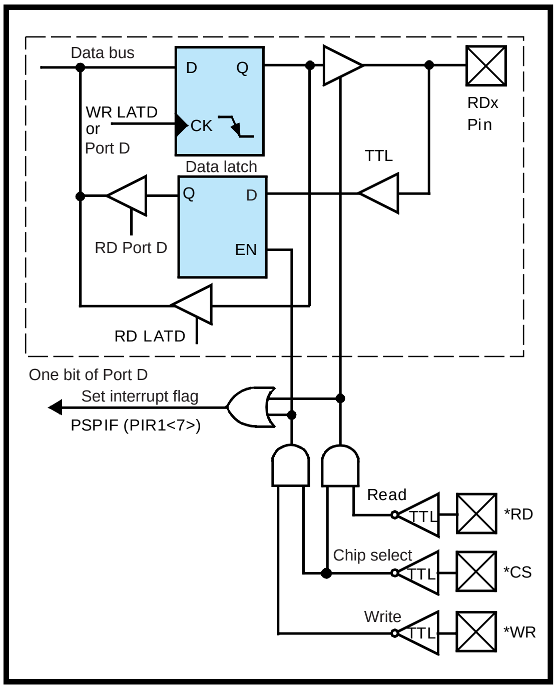


Figure 4—*Echoing its original Peripheral Interface Controller roots, the parallel slave interface offers an easy way to connect to, and offload I/O drudgery from, a host CPU.*

do little more than wait for the conversion to complete. In fact, if the MCU is running faster than 1 MHz, the datasheet indicates it should be put to sleep during conversions using the RC oscillator, lest accuracy suffer.

Finally, there's an option to configure Timer 2 and the CCP module to trigger repeated A/D conversions. The housekeeping is handled in hardware and the A/D result register can be read at your convenience.

ONWARD & UPWARD

What's next? Be watching for '18C*xxx*s with more memory, flash memory (Microchip has adopted a migratable memory strategy that makes all chips available in ROM, OTP, and flash), and bigger packages with more and fancier I/O such as USB and CAN bus.

With plenty of speed, memory, and peripherals, the '18C*xxx* is a far cry from the minimalist PIC of the past. The features, along with $5+ pricing, expand Microchip beyond its low-cost niche and into the high end of the 8-bit MCU biz.

Lest you fear that Microchip is abandoning its roots, don't forget that when it comes to silicon, today's high end is tomorrow's low end. By the way, for those of you who want to be part of "tomorrow," Microchip is sponsoring the Internet PIC 2000 design contest (www.circuitcellar.com/online). ■

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for more than ten years. You may reach him by e-mail at tom.cantrell@circuitcellar.com, by telephone at (510) 657-0264, or by fax at (510) 657-5441.

SOURCE

PIC18C*xxx*
Microchip Technology, Inc.
(888) 628-6247
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

CIRCUIT CELLAR Test Your EQ

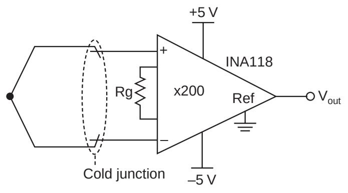
Problem 1—You have just encountered some code that was written to time the execution speed of a function. The resulting speed measurement varies widely from one execution to the next. What is the problem?

```
void func_under_test(void);

speedtest()
{
  clock_t start, stop, total;
  int i;

  total = 0;
  for (i=0; i<1000; i++)
  {
    start = clock();
    func_under_test();
    stop = clock();
    total += stop - start;
  }
printf("execution time = %f\n", total / (float)1000);
}
```

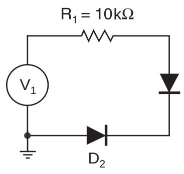
Problem 2—You have been assigned the task of programming a robotic mouse to escape a maze. The maze is guaranteed not to have any islands. Can you describe a general algorithm that will determine which route to take without keeping track of where you have been so far?

Problem 3—Your coworker builds the thermocouple interface shown in the figure. The INA118 is an instrumentation amplifier suitable for amplifying small signals, such as those from a thermocouple. A typical input impedance on the INA118 is about 10 GΩ.

When power is first applied to the circuit, the output is proportional to temperature differential between the hot and cold junctions of the thermocouple. However, after a few minutes, the output saturates to one of the rails. If left alone, sometimes the circuit seems to begin to function again; sometimes, it remains saturated until power is disconnected and then reapplied. What's going on?



Problem 4—Given the circuit in the figure, and assuming ideal diodes, what is the voltage imposed across the resistor? (An ideal diode has 0 Ω of resistance when forward biased and an infinite impedance when reverse biased.)

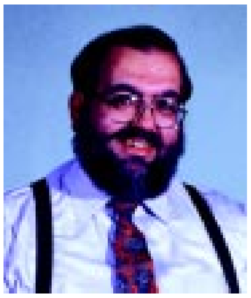


$$V_1 = 10 \cdot \sin\left(\omega_0 \cdot t + \frac{\pi}{3}\right)$$
$$\omega_0 = 2 \cdot \pi \cdot f_0$$
$$f_0 = 60 \cdot Hz$$

Have you forgotten the basics since getting that management job? Did your university ignore real circuits and tell you that HTML was the only path to fortune? Each month, Test Your EQ presents some basic engineering problems for you to test your engineering quotient. What's *your* EQ? The answers are posted at www.circuitcellar.com along with past quizzes and corrections. Questions and answers are provided by Ray Payne of Under Control, Benjamin Day of Schweitzer Engineering Laboratories, and Bob Perrin of Z-World. You may contact the quizmasters at eq@circuitcellar.com.

PRIORITY INTERRUPT

A Heavy Experience in List Management



I've spent the last few editorials describing my visions for *Circuit Cellar*'s future. If nothing else, you should recognize that I'm a pretty conservative guy when it comes to the magazine. And I'm not just conservative with regard to content, I'm also rather conservative when it comes to management—especially when it comes to our subscriber list. The nightmare for all publishers is that something inadvertently happens to that list.

Regardless of my in-house management policies, publishing a print magazine involves precisely coordinating a number of subcontracted services. The magazine address is Vernon, CT. That's where the entire contents of the magazine, things like editorial and advertising, are put together as digital files.

Once a month these files are sent to Dartmouth Press in New Hampshire where they physically print *Circuit Cellar* along with 40 or 50 other magazines. (If you've never seen a 100′ four-color web press in action, it's worth a trip to a printer.) Before they roll the presses, they look at a physical print order from us that says how many issues to print and where to mail them after they are printed. The magnetic tape containing all the subscriber names for the issue comes from our subscription fulfillment house in Philadelphia.

Contrary to what you might have thought, and in spite of the personal care you get on subscription questions from Rose Mansella here in our office, only the largest (or smallest) magazines can afford to handle their own subscriber lists. The fulfillment service has the people who open the envelopes with your renewals and key in all the information and address changes. Like the printer, a fulfillment service handles multiple magazines. When it works, it's a great system.

According to the fulfillment service, they screwed it up for the first time in 35 years in July (and I was the lucky winner). The person shipping the magnetic tapes put the subscriber tape for *Heavy Metal* magazine in the box to Dartmouth and sent our list to *Heavy Metal*'s printer. Consequently, at least 15,000 *Circuit Cellar* subscribers received the latest issue of *Heavy Metal*!

I found out about all this just like most of you did. I went home and my wife said, "You'll never guess what I got in the mail today." Needless to say, I was horrified when I finally traced it back to our own list!

The saga didn't end there. Just as Dartmouth was about to mail August's *Circuit Cellar* to the list of subscribers on the fulfillment tape they had be given, a smart lady up there looked at our print order and noticed that something didn't add up. They had printed a lot more magazines than there were names on the tape. Dartmouth called the fulfillment house and asked for another tape. The second time around, the right tape got to New Hampshire. If Dartmouth hadn't been on the ball, a good portion of our August issue would have ended up in the hands of *Heavy Metal* readers. (I wonder if they would view *Circuit Cellar* with a similar horror?)

The fact that we didn't lose our August issue alleviates some of the pain in this fiasco. Still, I feel the need to apologize to all of you. It was not a case of misappropriating our subscriber list, but it was a mistake (big). I just wish it had been a magazine like *US News* or *Scientific American* instead of *Heavy Metal*.

The primary reason for my talking about the exact details of this screw-up is because I don't want anyone to think that I would ever rent our list to such an inappropriate and tasteless magazine as *Heavy Metal*.

Yes, I occasionally rent the subscriber list. However, I assure you that I always personally review what they want to mail to you and I have my home address on that list (as a hidden seed) to verify it. I value my relationship with all of you, and I respect that subscribing to the magazine also involves a certain level of propriety with your name and address. I appreciate that in your responses to this situation no one accused me of making a fast buck with the list. The events in question were a mistake. We will endeavor to prevent it from happening again.

Steve

steve.ciarcia@circuitcellar.com