

www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#112 NOVEMBER 1999

ANALOG TECHNIQUES

Dual-Slope ADC Techniques

Build an Autoranging Frequency Counter

HDTV—HD Formats and
Signal Transportation

Protocol Implementation
Using IrDA Solutions



CIRCUIT CELLAR **ONLINE**

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

— FEATURES —

A Switcher for Many Reasons

Lawrence Foltzer

Need a quick solution for a variable high-voltage application? Lawrence shows us that the answer may be as simple as making a variable burst length, fixed duty-cycle switching power supply driven by an RC-clocked PIC microcontroller.

Embedded Systems *n*-Formation

Implementing an *n*-Tiered Client-Server Architecture

Mark Taylor

In corporate environments, who knows what depends on who knows who, and who knows who depends on who who is. As Mark shows us, the *n*-tiered architecture was designed to make sure the right people have access to the information they need.

Testing 1, 2

Part 4: Immunity—Not for Circuitry

George Novacek

Your design needs to be rock solid regardless of what frequency is thrown at it—and that's a lot these days with the proliferation of cellular phones, radar, and microwave technology. And, your design needs to keep its radiation of frequency to itself as well. Check in with George to find out what tests the lab will throw at your design to make sure it's ready for the real world.

Resource Links

- [The 80186](#)
- [RS-485 Multidrop Networking](#)

Benjamin Day

— COLUMNS —

Considering the Details

I/O for Embedded Controllers—Part 2: Analog I/O

Bob Perrin

Embedded systems are used in so many different applications, it's impossible to cover all possible analog I/O requirements. In Part 2 of this series on I/O, Bob offers a few circuits and components that have proven adequate for many applications in the past.

Lessons from the Trenches

Timing is Everything

George Martin

If you've ever wanted to choke the person who made up a project design schedule, or if you are the one responsible for making the schedules, you might want to listen to what George has to say about the importance and benefits of making a realistic, achievable, and practical plan for bringing your design to completion.

Silicon Update Online

16-Bits or Bust

Tom Cantrell

If 8-bit chips are the compact pickup truck and 32-bit chips are the Corvettes, what does that make the 16-bit chips? Tom does some homework and hopes to find out exactly where 16-bit chips fit in the MCU market.

WWW.CIRCUITCELLAR.COM/ONLINE

Table of Contents for October 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

- [Fuel Cells and Radioisotope Heater Units](#)
 - [Joint Test Action Group \(JTAG\) IEEE 1149.1 and IEEE 1149.4](#)
- Bob Paddock

Test Your EQ

8 Additional Questions

- 12** **Build a MIDI Sustain Pedal**
Bill Dudley
- 20** **Working with a Dual-Slope ADC**
Richard Lao
- 26** **Embedded Living**
Tuning into the HA Channel
Mike Baptiste
- 36** **What's the Count?**
Build an AVR-Controlled Frequency Counter
Stuart Ball
- 60** **IrDA Technology**
Part 2: Protocol Layers
Hari Ramachandran
- 66**  **MicroSeries**
High-Definition TV
Part 1: Video Formats and Transport
Mark Balch
- 74**  **From the Bench**
Without Acceleration
Part 1: All We Have Left is Velocity
Jeff Bachiochi
- 78**  **Silicon Update**
LPC—The Little Processor that Could
Tom Cantrell

Task Manager Elizabeth Laurençot	6
Household Variable = Steady Work	
New Product News edited by Harv Weiner	8
Test Your EQ	83
Advertiser's Index December Preview	95
Priority Interrupt Steve Ciarcia Another Typical Trip	96

INSIDE ISSUE 112

EMBEDDED PC

- 44** **Nouveau PC**
edited by Harv Weiner
- 46** RPC **Real-Time PC**
Serial Port Interfacing
Ingo Cyliax
- 52** APC **Applied PCs**
Sending a DOS Stamp Airmail
Fred Eady

TASK MANAGER

Household Variable = Steady Work



The theme of this issue—*analog techniques*—should bring to mind projects that have to do with measuring a continuous physical variable. Think voltage. Think pressure. Think TV.

Huh? One “continuous physical variable,” in my household at least, is the television—although as far as we’re concerned, the primary variable concerns who has control of the remote and which channel is currently being broadcast into our living room.

And as for how this variable may change over time, I imagine that a decade from now, we’ll still spend some of our time watching the tube. But of course, now we’re being told that it’s not going to be the same old TV after all. As our *MicroSeries* columnist this month, Mark Balch, tells us, there’s no doubt that high-definition television is coming to stay.

There’s a bit of time before the FCC-mandated curtain falls in 2006, and to tell the truth, I haven’t been all that interested in high-definition technology so far. In fact, I’ve been wondering, what’s all the fuss? So we get a wider view of the TV—so what? Just what my family needs: more screen to stare at, right?

However, I decided to investigate a bit further what else HDTV has in store for us. After reviewing some of the information in one of the online newsletters devoted to HDTV, I’m starting to get a clearer picture of the benefits of the technology. I visited <http://web-star.com/hdtv>, but a quick Internet search will show you how *many* of these newsletters are out there!

One exciting area is the high-quality imaging necessary for flight simulators. Apparently, top-end flight simulators have to deal with compatibility of line rates and aspect ratio, the overlaying/compositing of multiple images to obtain realistic effects, and splitting the output of image generators to feed several display devices simultaneously.

On the educational and cultural fronts, you’ll find high-definition devices at museums and schools as well. For example, museums will offer wall-size video displays and kiosks showing slide-quality images of various artworks.

As might concern you more personally, if you have a health issue to resolve, consider the impact of high-resolution systems on medical imaging. According to Dr. Robert Brecht at the University of Texas Department of Biomedical Communications, “Subjects that make NTSC an insufficient visual medium are pathology, radiology (they don’t like anything under 1000 lines), microanatomy, telediagnosis, and CAD-CAM.”

It sure looks like a lot of these high-definition imaging systems are part of various embedded systems. So, after you get more HDTV info from Mark’s series, I hope you’ll be motivated to hit the power-off switch on your own remote control and spend your time designing some cost-effective, viable, high-resolution devices for our everyday lives.

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

SENIOR TECHNICAL EDITORS

Steve Meyst

CHIEF FINANCIAL OFFICER

Jeannette Ciarcia

TECHNICAL EDITORS

Michael Palumbo Rob Walker

CUSTOMER SERVICE

Elaine Johnston

WEST COAST EDITOR

Tom Cantrell

ART DIRECTOR

KC Zienka

CONTRIBUTING EDITORS

Mike Baptiste Ingo Cyliax Fred Eady
George Martin Bob Perrin

GRAPHIC DESIGNER

Jessica Nutt

NEW PRODUCTS EDITOR

Harv Weiner

ENGINEERING STAFF

Jeff Bachiochi
Steve Bedford
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.

Circuit Cellar® makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors. Furthermore, because of possible variation in the quality and condition of materials and workmanship of reader-assembled projects, Circuit Cellar® disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in Circuit Cellar®. Entire contents copyright © 1999 by Circuit Cellar Incorporated. All rights reserved. Circuit Cellar and Circuit Cellar INK are registered trademarks of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

NEW PRODUCT NEWS

Edited by Harv Weiner



EVALUATION KIT FOR GPS RECEIVER MODULES

The **SGM5600EK** is a kit for evaluating GPS receiver modules. It includes the SiGEM SGM3900 low-noise GPS active antenna, 9-V AC adapter, automobile power plug, 7.5' serial cable, CD mapping software, and SiGEM's SGM5606PS GPS receiver. There is an NMEA serial port, a DGPS serial port, a satellite lock indicator (LED), and a hardware reset button on the front panel. A switchable 3.3- or 5-V DC feed to an active antenna is also included. The antenna has a noise figure of less than 1 dB and consumes less than 20 mW.

DGPS corrections can also be provided to the GPS receiver internally, by mounting a suitable DGPS card, or externally, by connecting to the serial port through an RS-232 interface. Extra internal board space permits customization. An 8-15-V input supply range is available.

The GPS receiver module communicates with computers via NMEA 0183-formatted messages. MapSite software is bundled with the evaluation kit and provides an accurate picture of GPS position, track, waypoints, and the route between them. It can also create maps from compatible scanned maps and works well with commercially available digital raster maps. The software can be used to record real-time position information and track from the SiGEM GPS receiver. Sentences are in GCA, GSA, GSV, and VTG NMEA formats. The SiGEM modules also support RTCA-SC159, WAAS, and EGNOS DGPS data formats. ToolKit software is also provided for integrating GPS functions into Visual Basic applications.

The SGM5600EK evaluation kit sells for **\$299**, or for **\$379** including MapSite and ToolKit.

SiGEM, Inc.
(613) 271-1601 • Fax: (613) 271-1896
www.sigem.ca

DIFFERENTIAL AMPLIFIER

The **DFA 5** is a low-voltage differential amplifier for test and measurement applications. Gain settings from 1 to 1000 are switch selectable and have an accuracy of 1%. The unit may be used as either a differential mode or a single-ended mode amplifier. With common-mode noise rejection that exceeds 100 dB, the DFA 5 makes low-voltage measurements straightforward. The unit is suitable for use with oscilloscopes and other common test equipment.

The DFA 5 is designed to amplify differential signals ranging from several volts down to microvolts. Maximum frequency depends on the gain setting, and ranges from 20 kHz at a gain of 1000 to over 1 MHz at unity gain. The unit allows for both AC and DC coupling, with an AC mode low-frequency cutoff of 10 Hz using nonattenuating probes.

The unit is small, lightweight, and low in cost. It can run for days on its internal battery or be powered by an external power source.

The DFA 5 sells for **\$129**.

Allison Technology Corp.
(281) 239-8500
Fax: (281) 239-8006
www.atcweb.com



NEW PRODUCT NEWS

68HC908-BASED SINGLE-BOARD COMPUTER

The **CP-908** is a single-board computer based on the 68HC908 microcontroller. The 68HC908 has 20 KB of on-chip flash memory for program storage and a variety of control-related resources. These include an eight-channel, 8-bit A/D converter, eight keyboard interrupt inputs, eight general-purpose I/O lines, built-in monitor ROM, SCI (asynchronous) serial port, SPI (synchronous) serial port, timer interface module, and 512 bytes of RAM.

A unique feature of the 68HC908 is that its flash memory is in-system programmable. Because user software can erase, write, and read the flash memory, it can be programmed in the standard user mode. However, a special Monitor mode, which enables the user to perform various low-level operations on the 68HC908 without executing user code, is also available. The CP-908 supports both User mode and Monitor mode in-system programming by manual switch selection.

The CP-908 has onboard voltage regulation, power up reset logic with manual reset switch, two 26-pin



header connectors bringing out all 68HC908 lines, a 10-pin header for connecting to the host PC, and A/D input buffering circuitry. The board measures 4" × 2.5".

The CP-908 sells for **\$149**.

Allen Systems
(614) 488-7122
Fax: (614) 488-7122
members.aol.com/allensys

NEW PRODUCT NEWS

ADAPTABLE OPERATOR INTERFACE

The **OP6100**, an operator interface designed for control-system communications, provides a functional, easy-to-use unit for entering commands, sending or receiving messages, or monitoring system functions. A 4×6 keypad with 24 tactile buttons enables quick and efficient data entry. The 4×20 LCD is easy to read from any angle. A backlit version is also available.

When the interface is connected to compatible Z-World controllers, programming is simplified with Z-World's integrated Dynamic C software. Dynamic C provides all software drivers necessary to scan keypads, display messages, and create graphics. A keypad overlay provides the capability to easily create custom keypad legends. This development system has device-specific libraries that provide powerful and easy-to-learn programs that reduce design efforts and costs.

The OP6100 can be customized for specific applications and interfaces easily with components such as other LCDs, keypads, and board controllers. A mounting kit, including NEMA-4 bezel and gasket, for flush-panel mounting is available for **\$60**. An optional enclosure ($5.30'' \times 6.83'' \times 2.32''$) is also available.

The OP6100 sells for **\$159**. Units with LED back-lighting are priced at **\$209**.

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



FEATURE ARTICLE

Bill Dudley

Build a MIDI Sustain Pedal

Bill set out to take care of the shortcomings of his wife's electronic keyboard by building a MIDI sustain pedal using a 68HC11 and C. Sit back and enjoy the performance as he gets the hardware and software in tune.



This was one of those projects that sounds like a good idea then quickly turns into spending so much time building the widget that you could buy ten commercially manufactured widgets for the money you didn't make while building the project. But, it was more fun than breaking rocks for a living.

My wife is a musician, a keyboard player. One of her recent acquisitions was a small, lightweight keyboard. This keyboard was cheap so it didn't provide for a sustain pedal.

Initially, Kate thought it would be OK, but after a while, she decided that a sustain pedal would be nice. That's when I spoke up.

"Sure honey, I can build you a sustain pedal. (pause) What's a sustain pedal, anyway?" And that's how this project started.

A sustain pedal, when pressed, removes the damping mechanism from the sound-producing mechanism, so a note will play until it decays away naturally. In a piano, the felt damping pads are lifted so the strings do not stop oscillating when the key is released.

MIDI has been the subject of several articles ("Digital Attenuators," *Circuit Cellar*

95; Jeff's MIDI series in *Circuit Cellar* 99-100), so I won't spend time describing it, except to say that MIDI stands for Musical Instrument Digital Interface and defines a protocol and physical medium for interconnecting musical input devices (keyboards, mostly) and musical output devices (namely, synthesizers).

MIDI uses current loop as the physical medium, the bit rate is 31,250 bps, and the messages consist of packets of (typically) 1-3 bytes.

I designed and built this project using some fairly high-powered tools and hardware to minimize development time. Obviously, if this design was going to be mass produced, you could spend more time in the design phase and cram the design into a tiny little microprocessor (e.g., a PIC).

Because I was making only one unit, I optimized for short design time. In this article, I describe the development process to show how designs are done in larger companies with reasonable tool budgets.

HARDWARE

First, I chose a microprocessor. I toyed with the idea of using an 8048 or a PIC, but since they have no internal serial port, I rejected them.

The MIDI data rate of 31,250 bps is high enough so the timing would be tight without a serial port to divide the interrupt rate by eight. I also had to deal with simultaneous input and output streams, so a bit-banging serial

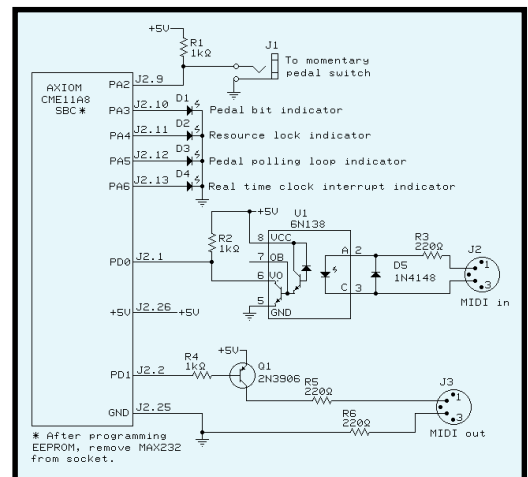


Figure 1—This is all the custom I/O you need to convert the 68HC11's serial port to MIDI. Because the MIDI interface is a current loop, the MAX232 supplied on the Axion SBC is removed when the MIDI interface is used.

Listing 1—The `midi()` task reads incoming MIDI messages and copies them out again, locking the MIDI-out resources when a MIDI message is partially processed. The header files whose names begin with a “c” (`clock.h`, etc.) are generated by the `RTXGEN` tool supplied with the `RTXC` kernel.

```
#include "hardware.h"
#if K4
#include <iok4.h>
#else
#include <i.o.h>
#endif

#include "rtxcapi.h"

#include "clock.h" /* CLKTICK */
#include "cres.h" /* SCIRES */
#include "cqueue.h"
#include "csema.h"
#include "lcd.h"

#define SELFTASK ((TASK)0)
#define TMINT ((TICKS)5000/CLKTICK)

static const char hex[] =
{ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
  'A', 'B', 'C', 'D', 'E', 'F' };

void xtoa(unsigned int x, int len, char *s) {
  s[len--] = '\0';
  while(len >= 0) {
    s[len--] = hex[x & 0x000f];
    x >>= 4;
  }
}

char buf[4];
extern char unsigned minbuf[], ihead, itail;
extern unsigned char porta;
static unsigned int inseq;

void midi(void)
{
  unsigned char true, ichar, havelock;
  true = 1;
  havelock = inseq = 0;
  init_lcd();
  gotoxy_lcd(1, 1);
  cputs_lcd("ON ");
  gotoxy_lcd(1, 3);
  cputs_lcd("OFF ");
  gotoxy_lcd(1, 1);
  while(true) {
    KS_wait(SCIISEMA); /* wait on input char */
    while (itail != ihead) {
      ichar = minbuf[itail++];
      switch (ichar & 0xf0) {
        case 0x90 :
          buf[0] = '\0';
          inseq = 3;
          gotoxy_lcd(4, 1);
          break;
        case 0x80 :
          buf[0] = '\0';
          inseq = 3;
          gotoxy_lcd(5, 3);
          break;
        case 0xa0 :
        case 0xb0 :
        case 0xe0 :
          inseq = 3;
          gotoxy_lcd(1, 2);
          goto Print;
        case 0xc0 :
        case 0xd0 :
          inseq = 2;
          gotoxy_lcd(1, 2);
          goto Print;
        case 0xf0 :
          switch(ichar) {
            case 0xf0 : /* system exclusive */
              inseq = 0xffff;

```

(continued)

port would have to handle interrupts at a 62-kHz rate.

I've had a lot of experience with the 6811 [1] and a little experience with the 8051, so I went with what I know. The only argument against the 6811 is that it's probably overkill for this project, but remember, I'm optimizing for development time, not cost.

I've used the 68HC11 C0 and K4 variants on professional jobs, but they have too much I/O capability for this project. Besides, mail-order 68HC11 SBCs always use the A or the E part. Good enough, we'll use the 68HC11A.

After searching the 'Net and looking through the ads in *Circuit Cellar*, I decided on Axiom's CME11A SBC. It has a 68HC11A microprocessor, with a MAX232 and DE-9 hung off the serial port, sockets for RAM/ROM/EEPROM, onboard power-supply regulator, plus an I/O decoder and header pins to connect an LCD module and keyboard.

Axiom also sells a development package that includes a wall-wart power supply, serial-port cable to connect to a PC, and software to compile and download programs to the onboard EEPROM.

This package is handy for quick demonstrations or prototypes. There's even a wire-wrap area on the CME-11A so you can add some custom I/O.

The Axiom board's built-in LCD port makes it convenient to attach an LCD module [2] that uses the Hitachi 44780 or equivalent LCD controller. This setup allows a convenient debug message display from your embedded system, which is useful if the target's only serial port is otherwise occupied.

TOOLS

When I'm building an embedded system, I always start in C. If I run out of room or real time, I'll drop back into assembler, but generally that isn't necessary. ROM is just too cheap.

(Whenever I've started an embedded project in industry, I've always designed in the current "popular" ROM size, with a way to expand to the "cutting edge" size ROMs. Every time, without exception, by the time the project hits production, the ROM I designed in is almost obsolete, and the "big" ROM is the default size,

and there's a much bigger one available that won't fit in the socket I designed. You'd think I'd have learned Moore's law by now.)

The C compiler I use is from Cosmic Software. It generates excellent code and comes with a reasonable subset of the standard C library.

There's a command-line version and a GUI version. I'm strictly a command-line kind of person. I just want to type in my C code, type `make`, and go get a Jolt.

`Make` is portable across all the systems I use (Berkeley Unix, Linux, DOS). So, I don't have to change the way I work even as I move from computer to computer during the day.

The next wonderful tool is an emulator. The crash-and-burn development cycle can get tedious, especially when nothing's running so you can't even run a debugger on the target.

An emulator lets you quickly see that your stack pointer is pointing to nowhere. Then, once you get the little sucker running, the emulator provides a profiler, instruction trace, complex breakpoints, and the ability to examine or change the registers or memory.

My Nohau emulator lives in a box the size of a toaster, communicates with a PC running DOS/Windows, and has a cable ending in a pod which plugs into the target in place of the microprocessor chip. Nohau-supplied software runs under Windows and lets you control the emulator. You can load code, set breakpoints, examine or change registers, and run your code.

This emulator has real-time full-speed trace of the target's execution and a performance analyzer to help figure out where the code is spending its time.

SOFTWARE

I decided this project needed an RTOS so I could have multiple tasks running independently without writing this complexity myself. I went with RTXC from Embedded Systems Products. It comes with complete source code and is customized for the particular microprocessor you're using.

The RTXC kernel supplies all needed (and even imagined) kernel services—that is, manipulation of mailboxes, semaphores, resource locks, queues, dynamic tasks, and so on.

Listing 1—continued

```
gotoxy_lcd(1, 4);
goto Print;
case 0xf2 : /* song pos */
inseq = 2;
gotoxy_lcd(1, 4);
goto Print;
case 0xf3 : /* song sel */
case 0xf6 : /* tune req */
case 0xf7 : /* end of sys excl */
case 0xf8 : /* timing clock */
case 0xfa : /* start */
case 0xfb : /* continue */
case 0xfc : /* stop */
case 0xfe : /* active sensing */
case 0xff : /* reset */
default : /* undefined: f1 f4 f5 f9 fd */
inseq = 1;
gotoxy_lcd(1, 4);
goto Print;
}
default :
Print:
xtoa((unsigned int)ichar, 2, buf);
buf[2] = ' ';
buf[3] = '\0';
break;
}
cputs_lcd(buf);
if(!havelock) {
KS_lockw(SCIRES);
havelock = 1;
}
porta |= SCIRESBIT;
PORTA = porta;
KS_enqueuew(SCIOQ, &ichar);
if(inseq) inseq--;
if(inseq == 0) {
porta &= ~SCIRESBIT;
PORTA = porta;
KS_unlock(SCIRES);
havelock = 0;
}
}
}
}
```

One of the nice parts about this tool set is that the three vendors—Cosmic, Nohau, and Embedded Systems Products—communicate with each other. For example, the emulator knows about the symbol table output by the compiler. The compiler works correctly with the kernel. And the kernel's internal tables and variables are understood by the emulator.

THE TASKS

This project needs at least two tasks. The `midi()` reads the incoming MIDI stream from the keyboard and copies it out to the next device in the chain (typically, a synthesizer). The `pdlpoll()` task polls the state of the I/O pin connected to the sustain pedal and transmits a pedal-state message when it detects a change of the pedal's state.

The `midi()` task listens to the MIDI input from the keyboard and copies whatever it sees to the MIDI output (serial output queue). More importantly, when `midi()` copies a MIDI message this way, it acquires a resource lock on the serial port queue, so it has exclusive access to this queue.

The `midi()` task in Listing 1 consists of an event loop that waits for a semaphore from the serial port interrupt handler, indicating that a character was received. A `while` loop empties the queue of received characters and feeds them into a `switch` statement, which classifies the MIDI messages by length, based on the first character and controls the display on the LCD module.

The integer variable `inseq` monitors the length of the MIDI messages. It is set to the length of the message

at the start of each MIDI message, and it is decremented for each subsequent character in that message.

When `midi()` sees the last byte of a MIDI message (signified by `inseq` reaching 0), it releases the resource lock on the serial-port queue so other tasks may use the serial port.

When `pd1poll()`, shown in Listing 2, regularly polls a pin on the 68HC-11A's parallel port A. When that bit changes state, it means the pedal was pressed or released.

The `pd1poll()` task then waits for the resource lock to be free, grabs the lock, stuffs the appropriate message into the serial port output queue (pedal up or down), and releases the lock.

If you're wondering why there needs to be a task to read MIDI in and copy it to the output, the answer lies in the fact that MIDI messages are multibyte packets. If the `pedalpoll()` task just blindly shoots out pedal-state messages, it will eventually send one in the middle of a MIDI "note on" or "note off" message from the keyboard. This will cause both messages

to become garbled, and the synthesizer will throw away one (or both) of the messages as corrupt.

So, the purpose of the `midi()` task is to read and understand the MIDI stream from the keyboard. That way, it knows when it's safe for `pedalpoll()` to send a pedal state message without interrupting a keyboard message.

`midi()` uses the resource lock on the serial-port queue to let `pd1poll()` know what times it is safe to send pedal state messages.

A third task—a device driver for the serial (MIDI) output—empties the serial port queue and sends the characters to the serial port transmit buffer register when an interrupt from the serial port transmitter indicates that the transmit buffer is empty and ready for another character. By using a semaphore, RTXC enables interrupt handlers to notify tasks of events.

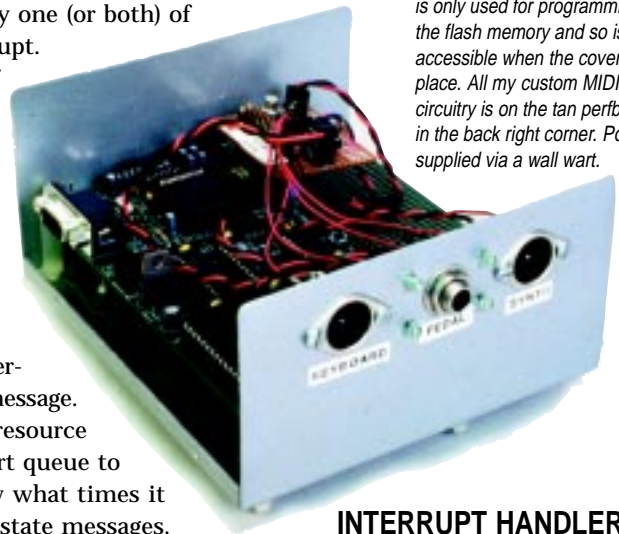


Photo 1—The DE-9 connector is only used for programming the flash memory and so is not accessible when the cover is in place. All my custom MIDI I/O circuitry is on the tan perfboard in the back right corner. Power is supplied via a wall wart.

INTERRUPT HANDLERS

As I mentioned, one of the interrupt handlers is devoted to the serial port interrupt. The 68HC11 has a whole slew of vectored interrupts, so it's easy to set up a different handler for every peripheral device that can generate an interrupt. When the built-in serial port of the 68HC11 receives or transmits a character, an interrupt is generated.

The handler reads the status register then responds to each of the possible interrupt sources. If a character is received, it is read and stored in a (global) buffer. And, a semaphore is

set to notify `midi()` that a new character is available to be read.

If the transmit buffer-empty interrupt has fired, a different semaphore is set to notify the serial output device

Listing 2—The `pd1poll()` task polls the pedal switch, and when it detects a change in pedal state, waits for the MIDI output resource lock before sending the appropriate MIDI pedal message.

```

#define SELFTASK ((TASK)0)
#include "hardware.h"

#if K4
#include <iok4.h>
#else
#include <io.h>
#endif

#include "rtxcapi.h"
#include "scidrv.h"
#include "csema.h" /* COMISEM, COMOSEM */
#include "cqueue.h" /* COMIQ, COMOQ */
#include "cres.h"
#include "serial.h"

#define POLLED 0

static const char downmsg[] = { 0xb0, 0x40, 0x7f };
static const char upmsg[] = { 0xb0, 0x40, 0x00 };
static unsigned char ped, lastped;
unsigned char porta;

/* poll for pedal change of state, when detected, send pedal message
 * (with resource lock to prevent message scrambling). */
void pd1poll(void)
{
    unsigned char i;
    unsigned char pedalsex;
    #if K4
        DDRA = 0x78; /* make it look like the port on a non-K4 */
    #endif
    porta = 0x60; /* guess if switch is N.C. or N.O. */
    lastped = PORTA & PEDALBIT;
    pedalsex = PEDALBIT & lastped; /* look at initial state of pedal */
    if(lastped) porta &= ~PLEDBIT;
    else porta |= PLEDBIT;
    PORTA = porta;
    for (;;) {
        KS_delay (SELFTASK, 2);
        porta ^= PDLPOLLBIT;
        PORTA = porta;
        ped = PORTA & PEDALBIT;
    #if POLLED
        monitor();
    #endif
    #endif
    if(lastped != ped) {
        if(ped) porta &= ~PLEDBIT;
        else porta |= PLEDBIT;
        lastped = ped;
        ped ^= pedalsex; /* correct for N.C. or N.O. pedal */
        KS_lockw(SCIRES);
        porta |= SCIRESBIT;
        PORTA = porta;
        for(i = 0 ; i < 3 ; i++) {
            KS_enqueuew(OQ, (ped) ? &downmsg[i] : &upmsg[i]);
        }
        porta &= ~SCIRESBIT;
        KS_unlock(SCIRES);
        PORTA = porta;
    }
}
}

```

driver task that the serial port hardware is ready for another character.

The second interrupt handler runs on the real-time clock interrupt of the 68HC11. This interrupt is scheduled to fire every 5 ms or so, and is used as a heartbeat by the RTXC kernel.

The kernel gets control at every real-time clock interrupt, when it checks to see if any task with a higher priority than the currently running task has become available. If so, a task switch is performed. Otherwise, the kernel returns control to the current task.

RAMPANT FEATURE-ITIS

Because a pedal is just a momentary switch, some keyboard manufacturers use normally closed switches and others use normally open switches.

My MIDI sustain pedal box would be ambidextrous. So, `pd1poll()` checks the initial state of the pedal when the software starts up. If the pedal pin is at a high logic level, the pedal is assumed to be normally open. Otherwise, it is assumed normally closed.

This information is then used when generating the pedal state messages, to generate the correct (pedal up or down) message regardless of which pedal you plug into the device.

Besides the two five-pin DIN connectors for MIDI in and out, and the ¼ phone jack for the pedal connection, four LEDs are mounted on the front panel. These are driven by software events so I can monitor the unit's health. They started out as debugging aids, but they looked so pretty when running that I left them on the front panel of the finished device.

One LED toggles when the real-time clock interrupt fires. Another one shows the state of the pedal contact. A third shows the resource lock activity on the serial output queue. The fourth toggles at the frequency of the `pd1poll()` polling loop.

INTERESTING COINCIDENCE

I had the project just about done when a friend showed me an article entitled "PIC MIDI Sustain Pedal" [3]. Naturally, I was intrigued.

This article solves the easier problem by generating MIDI pedal state messages. However, it assumes the

keyboard and synthesizer are in one unit so they don't need to insert the pedal messages into the keyboard's MIDI output stream.

This application is simple enough for a PIC. It only has to poll the pedal contact bit and grind out serial messages—not read any MIDI stream so it can coordinate with it.

Although my MIDI box wasn't a cost-effective project to build, it certainly was entertaining and allowed me to showcase some fine tools. ☒

Bill Dudley is a programmer for Monmouth Internet. He has designed embedded systems for the likes of AT&T Bell Labs and a whole assortment of much smaller companies. When not hacking around on computers he can sometimes be found riding one of his motorcycles. You may reach him at dud@casano.com.

REFERENCES

- [1] Motorola, *M68HC11 Reference Manual M68HC11RM/A*, Rev. 3, 1991.
- [2] Optrex, *LCD module*, Datasheet.
- [3] R. Penfold, "PIC MIDI Sustain Pedal," *Everyday Practical Electronics*, Feb. 1999.

SOURCES

68HC11A SBC

Axiom Manufacturing
(972) 994-9676
Fax (972) 994-9170
www.axman.com

68HC11 emulator

Nohau Corp.
(408) 866-1820
Fax (408) 378-7869
www.nohau.com

68HC11 C compiler

Cosmic Software
(781) 932-2556
Fax: (781) 932-2557
www.cosmic-software.com

RTXC kernel

Embedded Systems Products, Inc.
(800) 525-4302
(281) 561-9990
Fax: (281) 561-9980
www.rtxc.com

FEATURE ARTICLE

Richard Lao

Working with a Dual-Slope ADC

With its good noise immunity and notable filtering action over the digitization interval, a dual-slope ADC was Richard's first choice when designing a prototype of a system to measure stresses in a beam. And there's more to the story than that....



Working in the R&D lab and being involved in product development means wearing many hats in the process of producing a concept prototype. In such an environment, the prototype must be developed rapidly to demonstrate the feasibility of the design concept.

Sensors must be interfaced to analog signal-conditioning circuitry. Those circuits must be designed, and often the sensors have to be designed as well. The analog signals must be multiplexed, digitized, and processed, and ultimately some useful information must be displayed on, say, an LCD. It's the digitization that interests us here.

For example, consider strain gauges arrayed in a Wheatstone bridge configuration to measure stresses in a beam. The bridge voltage must be amplified, filtered, and then sent to an ADC.

For good noise immunity with such DC or slowly varying voltage signals, you can use a dual-slope or sigma-delta ADC. Here, I consider the dual-slope ADC, which is noted for its filtering action over the digitization interval.

PLAYING WITH PROTOTYPES

A prototype represents a product to be manufactured, and so, must have the same features—inexpensive parts, and as few of them as possible. Like

most products, this one needed an embedded microcontroller. In that sense, I already had part of the ADC. I merely needed to design a front end.

Because I was using a Microchip PIC16F84 (actually a Micromint PicStic-1) to handle housekeeping in the prototype and needed to rapidly prototype my design, I picked up my Microchip apps manual to search for some dual-slope ADC circuit and software boilerplate. Why reinvent the wheel?

Surprise! There were no adequate app notes for what I needed—a 13-bit dual-slope ADC. So, I resurrected some of my old designs and started the process of laying down a flowchart and coding. I didn't even have to use interrupts. By polling twice in each ADC loop, I could minimize errors resulting from polling down to 1 bit.

I elected to use a PicStic-1 for good reason. Although the PicStic-1 is relatively expensive compared to a "bare" '16F84, the price is offset by the convenience of its package (a 14-pin SIP).

My PC sits on my desk with my EEPROM programmer, and my bench is in the lab. The prototype circuit on the bench was a solderless breadboard. I could rapidly insert or remove the PicStic-1 for another programming iteration, which was easier than plugging in or popping out an 18-pin DIP.

The EEPROM programmer was an Epic Plus Pocket PICmicro Programmer from microEngineering Labs. This and their Epic PicStic adapter socket (ZIF) were convenient for burning-in and erasing PicStics for repeated program changes. (Of course, there was more to the program than the ADC routine.)

CIRCUIT DESCRIPTION

The ADC circuit in Figure 1 consists of a power supply, the PicStic-1, a CMOS 4051 mux, an (inverting) integrator, comparator, input buffer op-amps, LCD, as well as a few resistors, capacitors, and diodes. The power supply consists of two 9-V batteries with linear voltage regulators (LM78-L05ACZ and 79L05ACP) providing +5 and -5 V respectively for the ICs.

The circuit can be modified to work on a single supply. But for my application, I needed the negative voltage for other purposes. The '4051 mux (U3)

functions both as a multiplexer for input signals and as an integral part of the ADC front end.

The '4051 is an eight-channel multiplexer. One channel is used for the ADC reference voltage, leaving seven to measure various input voltages.

In Figure 1, only two input channels (x6 and x7) and the reference (x5) are used. The other inputs are grounded and the '4051 control line C (pin 9) is tied high. To use all of the channels to measure input voltages, control line C would have to be routed to the PicStic, as control lines A and B currently are, and the program would have to be slightly modified.

The PicStic-1 has a 4-MHz clock frequency, requiring four clock cycles per instruction (i.e., 1 μs/inst). The counting loop in the assembly-language program is polled every five instructions (5 μs). Thus, counting to 8192 requires 8192 × 5 μs, or 40.96 ms.

I chose an LF412CN dual op-amp for the integrator (U4A) and comparator (U4B) by default. I keep this op-amp around the bench for R&D purposes and use it like other folks use '741s. Until I settle the op-amp specs "in concrete," the '412 serves nicely. But, feel free to use another chip.

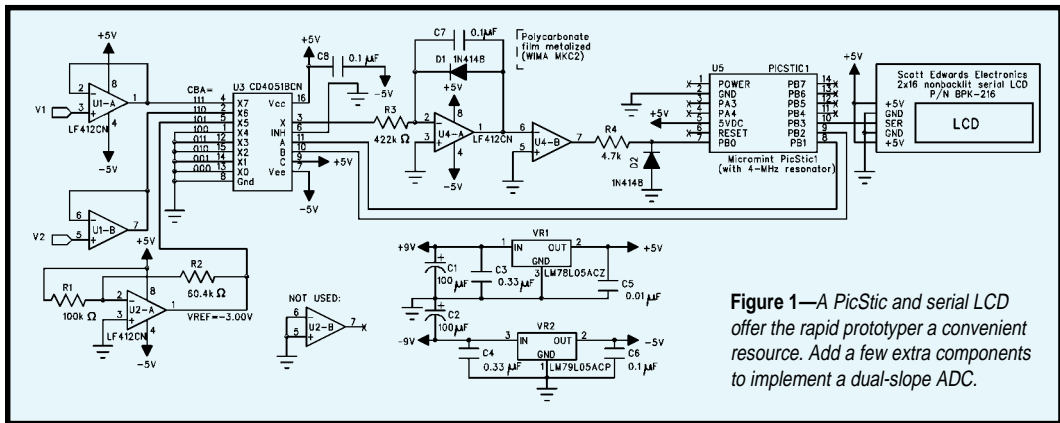


Figure 1—A PicStic and serial LCD offer the rapid prototyper a convenient resource. Add a few extra components to implement a dual-slope ADC.

For a quick room-temperature benchtop setup in R&D, a 1N4148 signal diode is satisfactory for D1, which is the limiter diode on the integrator. However, the 1N4148 has a relatively large reverse current over temperature, and for permanent design as an integrator limiter, it isn't a good choice. A low-leakage diode, like Digi-Key's FLLD258CT-ND would be just what the doctor ordered (I_R max. = 3 nA, I_R typ. < 25 pA)

Other circuit variations for dual-slope ADCs may include an analog switch to discharge the integrating capacitor during part of the measurement cycle.

DUAL-SLOPE REVIEW

A dual-slope ADC produces a digital count that's proportional to the positive voltage being measured once every conversion cycle. For example:

$$\frac{V_{meas}}{V_{ref}} = \frac{n}{N}$$

where V_{meas} is the (positive) input voltage being measured, n is its corresponding digital count, V_{ref} is the reference voltage (-3 V in this device), and $N = 8192$ (i.e., 2^{13}) is the digital count corresponding to $|V_{ref}|$.

What's more, V_{meas} is the average value of the voltage over the part of the conversion cycle during which it is being measured. That's why this kind of ADC is noise tolerant.

Random fluctuations of the input voltage during the measurement cycle are filtered out. The dual-slope ADCs in commercially available voltmeters are basically the same, except they have a network of internal switches to accommodate bipolar voltages.

There are three phases in the A/D conversion cycle. The first phase is zero phase (initialization). The integrator is connected to the -3-V reference voltage, which, in the absence of D1, causes the integrator's output to ramp up and ultimately reach positive saturation.

The integrator's output voltage is maintained (clamped) by diode D1 to one diode drop above zero for 40.96 ms, the time it takes the microcontroller to count to $2^{13} = 8192$. The time interval isn't critical and can be shorter.

I used the 8192 count because this count is used in the next phase. Eliminating D1 wouldn't defeat the ADC's operation but would consume a lot more time with no added benefit ramping up to and down from positive saturation.

The next phase is Phase 1 (signal integration). Pins 8 (PORTB1) and 9 (PORTB2) of the PicStic then address the '4051 mux (CMOS analog switch)

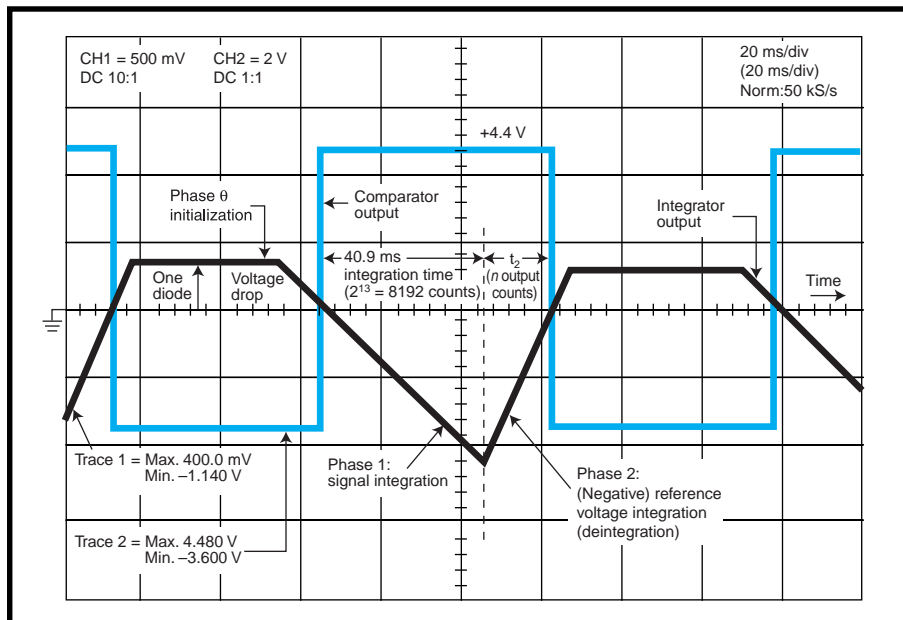


Figure 2—Integration count begins when the falling voltage ramp passes through zero and ends after 8192 counts. The deintegration count then begins, ending only when the rising voltage ramp again passes through zero.

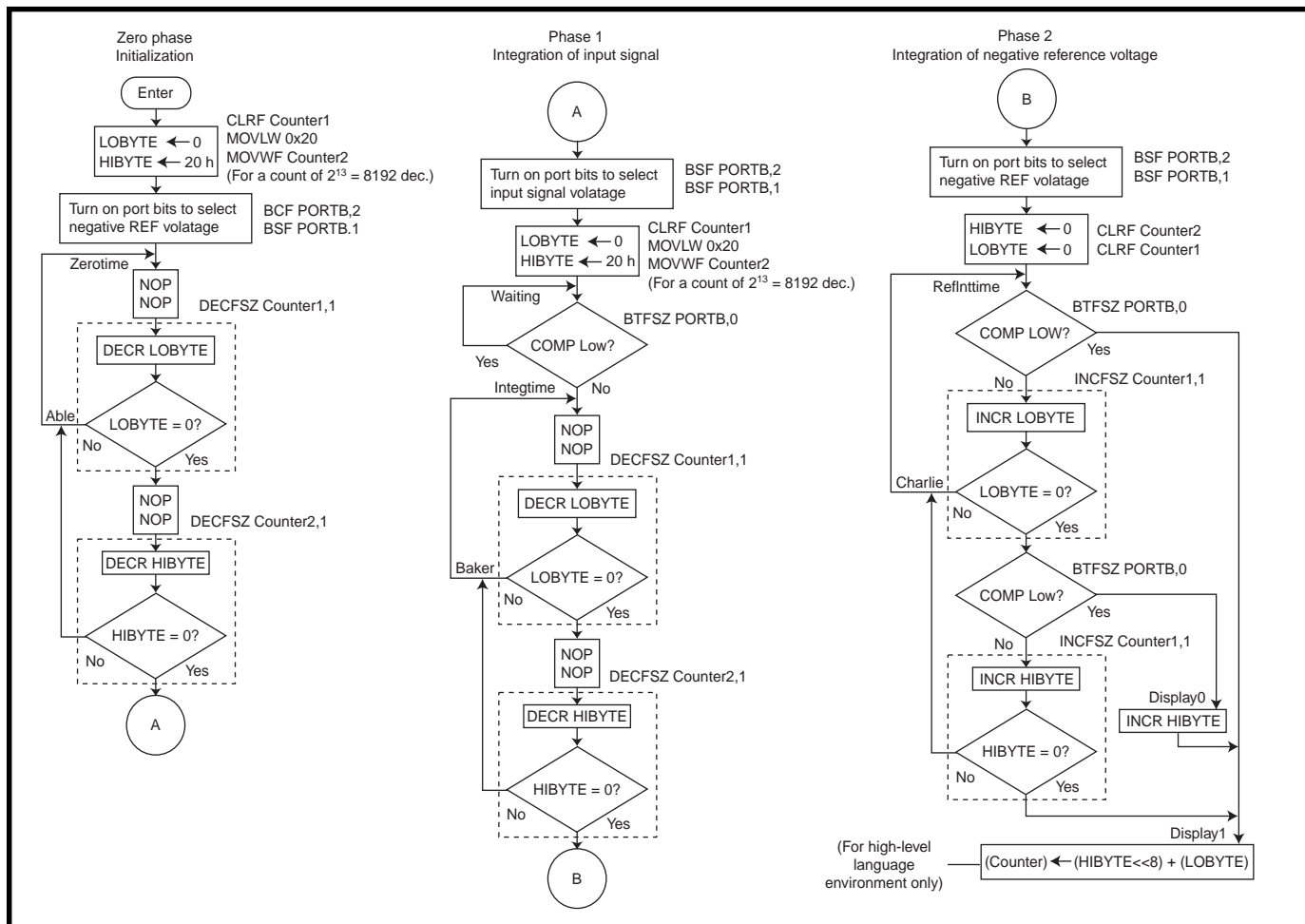


Figure 3—This flowchart is typical of microcontroller programming routines for dual-slope ADCs. The assembly-language mnemonics are for the PIC16F84. Interrupts are not used; rather, polling is employed in a loop.

to connect the (inverting) integrator to the positive input signal, causing the output to ramp down. When the ramp reaches 0 V and trips the comparator, the PicStic senses the zero crossing and starts counting.

The signal continues to be integrated for $N = 8192$ counts, the full count set in the program. Because the integration time is fixed for various input voltages, the slope of the ramp is not constant.

The third phase is Phase 2 (reference signal integration). The microcontroller switches the mux to the -3-V reference voltage, resets the counter to zero, and begins a new count.

For the positive voltages this ADC measures, the reference signal is of opposite polarity—a negative voltage. When the negative reference signal is (negatively) integrated, it ramps upward and essentially undoes the integration of the positive input signal (hence the alternate name for this phase, deintegration).

When the positive-going integrator ramp reaches 0 V, the comparator is tripped and the count (n) is stopped and stored. Figure 2 shows waveforms of integrator output voltage and comparator output voltage over time.

The total time for one conversion is about $46 + 40.96 + 40.96 = 128$ ms.

TRACKING AND RANGE

Suppose an input voltage to be measured is derived from a sensor operating from the circuit's 5-V supply (e.g., from a pot). Fluctuations on the 5-V supply are mirrored in fluctuations of the sensor output voltage and the ADC reference voltage. Given:

$$\frac{V_{\text{meas}}}{V_{\text{ref}}} = \frac{n}{N}$$

the ADC count will have immunity to power-supply fluctuations.

The voltage reference in this ADC is -3 V . In the `measure()` function in `Cellar1.c`, I could have written:

$$*adn = 256 \times \text{Counter2} + \text{Counter1};$$

Instead, I used some bit shifting:

$$*adn = \text{Counter2};$$

$$*adn = (*adn << 8) + \text{Counter1};$$

This shift-and-add stratagem is used to compact the code, but either way, the ADC will do its job. Note, however, that the `Cellar1.c` code formats the LCD to read counts up to four digits, then the LCD overflows. This can be changed to five digits by a minor change in the source code.

By contrast, in `Cellar2.bas`, I used $W2 = 256 \times B5 + B4$. Voltages above $+3\text{ V}$ can be measured, limited only by the power-supply voltage headroom.

DISPLAY

This project was conceived for a rapid prototyping scenario where development speed is essential. So, I

used a Scott Edwards 2 × 16 bit-serial LCD (BPK-216N), which contains a Hitachi HD44780 LCD controller.

These LCDs are a boon to development engineers fighting the clock. Because they require only three wires (+5V, GND, SERIN) for interfacing to the PicStic-1, I could concentrate on other matters besides LCD interfacing.

THE CONVERSION ALGORITHM

Figure 3 is a flowchart for the conversion routine. I wrote it in assembly language and embedded it, first in a BASIC program (Cellar2.bas) using microEngineering Lab's PicBasic Pro Compiler, and then in a C program (Cellar1.c) using Custom Computer Services' C cross-compiler, PCW.

PCW is a professional programming package that has a Windows IDE and includes two compilers, the PCB and PCM (for 12- and 14-bit opcodes, respectively). For the PicStic-1, I used PCM.

DISPLAY FORMAT

The LCD has different displays depending on which program is down-

loaded to the PicStic. With the BASIC program, the voltage on one channel (mux pin 2) will be measured. The count is shown on the first line. On the second line the voltage is computed using somewhat coarse (but satisfactory) integer arithmetic.

With the C program, the counts are displayed for two channels, with two samples averaged per channel—mux pin 4 on the first line, mux pin 2 on second line. A different number of samples may be averaged by changing the code in the appropriate places. ▣

Richard Lao has over 20 years' experience as a scientist and research engineer. He has developed electronic instruments, sensors, and systems, ranging from magnetometers and gyroscopes for oceanography and bore-hole navigation, to bio-tech. You may reach him at riclao@juno.com.

SOFTWARE

Source and hex code for this article in Basic and C are available via the *Circuit Cellar* web site.

SOURCES

BPK-216N

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

PicStic-1

Micromint, Inc.
(407) 262-0066
Fax: (407) 262-0069
www.micromint.com

Epic Plus programmer, PicBasic Pro Compiler, Epic PicStic adapter

microEngineering Labs, Inc.
(719) 520-5323
Fax: (719) 520-1867
www.melabs.com

C Compiler PCW

Custom Computer Services, Inc.
(414) 781-2794
Fax: (414) 781-3241
www.ccsinfo.com

Integrating capacitor

TAW Electronics, Inc.
(818) 846-3911
Fax: (818) 846-1194
www.tawelectronics.com

EMBEDDED LIVING

Mike Baptiste

Tuning into the HA Channel

A channel devoted to Home Automation may be a stretch, but in this month's column Mike shows us how to display HCS-II information via television with a little help from a PIC16C63A and an onscreen display module named BOB-II.



Some people say the ultimate home automation system is one you never need to interact with. It should run the day-to-day functions of your home on its own using sensors and programming.

I agree to a point. However, a good home automation system should also provide homeowners with information about their home, the status of its systems, or any urgent events or messages.

Getting this information usually means bringing up an application on a PC. People are spending more time in front of their PCs, but the television is used more often in most households.

Therefore, I think the TV is an ideal device for presenting vital home automation information. By adding an infrared interface, your home automation system can be controlled like any other A/V device.

I've always wanted to connect my HCS-II to my televisions (but I wasn't excited about designing the video section of the circuit). LCDs are nice for some places, but they usually aren't conveniently located and you can't see them from across a room.

I admit it—I'm a bit-head. I can hold my own when it comes to analog circuitry, but tinkering with NTSC video was a bit intimidating. Besides, I wanted it to be easy to build and I had pictures in my head of dozens of discrete parts making up the video portion.

I started looking around for chips that would handle onscreen displays without much external circuitry. I finally found a few, but they were surface mount and the spec sheets showed that controlling these OSD chips took a bit of register manipulation. The amount of code I needed to write was growing.

It became apparent that designing the NTSC circuitry wouldn't be the hardest part; sourcing the parts would. It was time to look for something off the shelf.

THIRD PARTY TO THE RESCUE

Recently, several companies have started selling complete onscreen display (OSD) modules that enable users to display text on a television. Communications are handled via a simple serial interface.

Because many are self-contained devices, I'd have one box with a custom controller connected to the third-party box. Not ideal, but if it worked, I'd be happy. But, I couldn't find one that met all of my requirements.

Since the HCS-II network runs at 9600 bps, I needed an OSD device that operated at 9600 bps or I'd have to buffer outgoing data as well as the incoming packets. Many OSD devices only operate at 1200 bps so HCS data arriving eight times faster would easily overwhelm the TV interface.

Video generation was another drawback of some OSD devices. Many require external video signals and can only generate text in monochrome;



Photo 1—The SIMM format of the BOB-II enables compact board design.

others don't allow direct cursor manipulation. One required sending the row and column at the beginning of each line displayed.

Given how most HCS-II LCD interfaces allow moving the cursor around at will, I needed to be able to move the cursor on demand when sending one line of text. I began to wonder if I'd have to brush up on my NTSC and search for obscure parts after all.

I had just about given up when I discovered a module called the BOB-II from Decade Engineering. The BOB-II is a self-contained OSD device on a 30-pin SIMM. It has a 9600-bps serial port and supports color text—two of my main requirements.

It also displays monochrome text over an existing video signal, generates its own video signal with multiple color backgrounds, overlays color text on internally generated backgrounds, and allows direct manipulation of the cursor position.

GETTING TO KNOW BOB

The BOB-II module packs a lot of functionality on a tiny SIMM circuit board. Photo 1 shows the BOB-II installed in the PIC-TV circuit board. The bulk of the video work is done by the SGS-Thomson STV5730A chip.

Using internal registers, the chip is controlled by an external microcontroller. The STV5730A also allows various configurations to be used for whatever application you have. Give the datasheet a read to see how this thing ticks. It's quite impressive.

On the BOB-II, an Atmel processor handles all the STV5730A control and external system communications. By using a 9600-bps serial interface, you can send various control commands (as well as normal text) to the BOB-II.

To make the BOB-II appealing to a wide market, the command set is RISC-like, simple, and straightforward. However, you can combine commands to make a more powerful interface. Table 1 shows the BOB-II command set. As you can see, it enables you to control most of the STV-5730A functionality.

Most commands are preceded by a { and consist of a command letter and

Command	Description
{A	Clears screen and moves cursor to home positions at x=0, y=0. Requires a 5-ms pause before sending more data.
{BE	Turn on display. Any text in display RAM is displayed.
{BD	Turn off text display. Existing text is maintained in display RAM and will reappear with a {BE command. Characters can be written to display RAM with display disabled.
{Cxxyy	Move cursor to xx,yy where xx and yy are two-digit numbers. No range check.
{Dn	Set character cell background color (0-7). Depends on {Kx command for background enable/disable. Local mode only.
{En	Set character color (0-7). Local mode only.
{Fn	Set screen color (0-7). Local mode only.
{GE	Blink enable. Subsequent characters blink.
{GD	Blink disable. Subsequent characters won't blink.
{HN	Use internal video levels.
{HX	Use external video levels set with potentiometers. (Not used on PIC-TV)
{In	Set character outline color (0-7). Local mode only.
{JE	Only cell backgrounds are colored with color set by {Dn. Note {KE must be sent to enable backgrounds.
{JD	Entire character grid is colored with {Dn background color. Noncharacter area is set to screen color. Makes a nice two-color screen.
{KE	Enable character backgrounds.
{KD	Disable character backgrounds
{MF	Local mode select. Forces BOB-II to generate video signal internally using current color settings.
{MM	Genlock/Overlay mode select. Monochrome text is overlaid on any existing video signal. If no video signal is present, the BOB-II remains in local mode.
{T. . <ESC>	Used to output special characters by their byte code. The codes are not ASCII!

Table 1—The BOB-II command set is compact making it ideal for machine to machine communication. There are no cursor control commands except for moving to a specific set of coordinates. Any other cursor control must be handled by a host processor (i.e., the PIC-TV).

sometimes a setting number or row, column pair. The compact command set ensures that you don't waste time sending long command names over the 9600-bps communications link and makes it easy to implement a more detailed command set on the HCS interface processor.

A number of normal characters like &, (,), %, and ! are absent from the BOB-II character set. Instead, many foreign-language characters are used, which makes the system more versatile.

It's clear why certain symbols were left off. The STV5730A chip is intended for use in self-contained devices like VCRs which don't use % and the like.

However, there are some neat symbols for use in a home automation system. There are arrows, blocks for graphing absolute values, and other video-related symbols.

Notice the character codes don't correspond to ASCII codes. If you send normal text in ASCII, the BOB-II converts it to the proper character code. However, if you want to use the special character function to select characters by byte codes, you must use the BOB-II character code.

IF ONLY IT WAS THAT EASY

With all the BOB-II can do, it might seem like adding it to the HCS-II would be a snap. Well, not quite.

I wanted the HCS-II TV interface to understand the same commands used for LCDs connected to LCD-Links or Answer MAN Jrs. This arrangement would make it easy to convert over to a TV display. It also means that people familiar with the LCD commands could use the TV interface without learning different commands.

Besides the command set, the TV interface had to handle RS-485 packet checksums, the HCS-II network protocol, ANSI cursor control, responding to HCS-II queries, and so on. It soon became clear that I would have to use an embedded processor to interface the BOB-II with the HCS-II.

The code I wrote for the PIC-DIO already handled the HCS-II networking, so it made sense to build the BOB-II interface on this platform. I could even use the '16C63A, which provides plenty of RAM for serial buffering, has a UART for serial interfacing, and enough ROM to handle

the numerous commands I planned to recognize. The HCS-II TV interface was now the PIC-TV.

Figure 1 shows the PIC-TV circuit. Most of the work is done in software. All the PIC-TV needed was an RS-485 interface, the main PIC controller, and a way to set the network address.

The network status LEDs do more than satisfy my love of blinky lights. You can tell instantly if you're getting HCS-II network data when you install it. And if you have network problems, you'll see network errors indicated by the flashing red LED.

Note that there is no voltage regulator in the circuit. The BOB-II uses a 78M05 regulator, which provides more than enough power for itself.

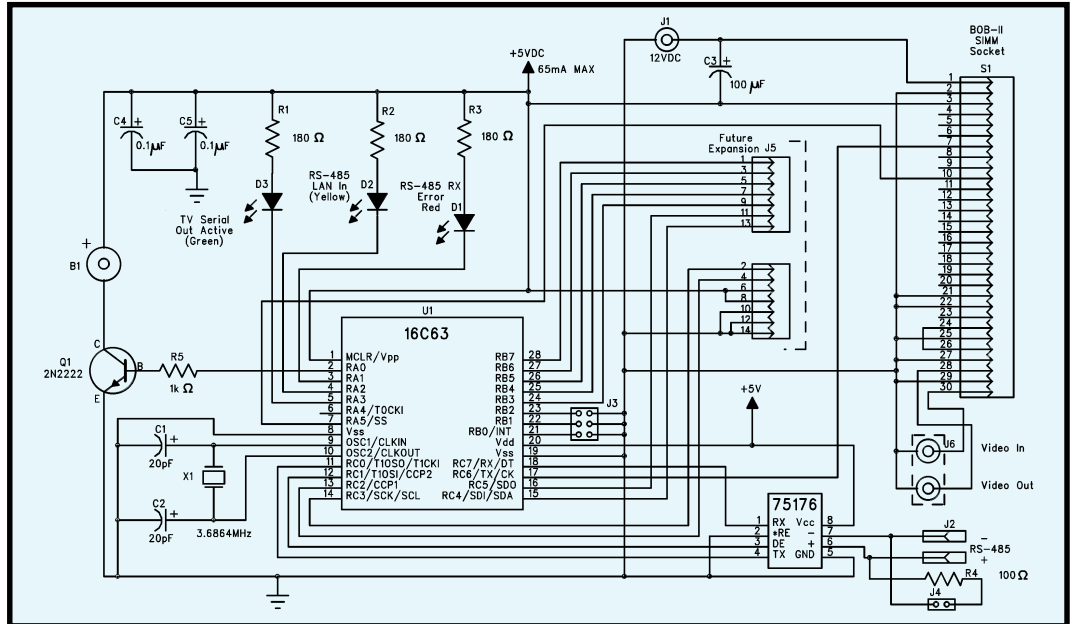


Figure 1—The PIC-TV hardware is fairly routine because most of the work is done in software. Note the lack of a voltage regulator—the BOB-II has one built-in.

The BOB-II brings the +5-V supply out to a SIMM pin for use by external circuitry, which is ideal for the PIC-TV's minimal power requirements.

The total current draw of the parts adds up to ~150 mA, but most of the

devices aren't on at the same time. Only one LED (15–20 mA) will ever be on at any given time.

The RS-485 chip draws ~40 mA during transmits, but that's not a problem because the LEDs are off

during transmissions. The beeper draws 30 mA, but it's rarely on.

The 78M05 can supply up to 0.5 A, but the surface-mount device doesn't dissipate heat well. So, the BOB-II specification states external current should be limited to 65 mA.

The PIC-TV exceeds this on a rare occasion, but in short bursts for LED blinks and packet transmits when the beeper is on. Testing revealed that the 78M05 gets warm at worst, so it allows me to drop voltage regulation from the circuit. Even in the case of an RS-485 network failure or a short on the +5-V supply, the 78M05 has an internal thermal shutdown.

One of the tricky parts of this interface is dealing with two asynchronous serial ports at the same time. The PIC-TV has to handle packets from the HCS-II whenever they're sent, yet be able to send data to the BOB-II at the same time.

I was having nightmares of a multi-tasking OS to handle both asynchronous ports, but then I cheated. The '16C63A has one hardware UART, which I planned to use for the HCS-II network. I really needed two so I can send characters to the UART buffer for the PIC-TV and let it worry about the timing and bit-banging while the receive buffer grabs HCS packets.

But since the HCS-II network is half-duplex, whenever the PIC-TV has to return data to the HCS-II, there's no incoming data to worry about. The BOB-II doesn't send any data we care about, so we can easily bit-bang the HCS responses without missing any incoming data or screwing up the bit timing with an interrupt. So, I split the UART in half.

The receive side receives HCS-II data and the transmit side sends data to the PIC-TV. This way, the PIC-TV can send data to the BOB-II without causing timing problems when it receives data from the HCS-II.

The HCS-II data is received via interrupt, while the PIC-TV sends data to the BOB-II in a loop. The '16C63A UART has a small buffer, allowing three characters to be received before it overflows. So, the PIC-TV could even preempt the interrupt for a little while if it needed to.

Although the PIC-TV doesn't return any usable data to the HCS-II, it appears to the HCS-II as an LCD-Link and must respond when queried to indicate that it's online. The PIC-TV has no keypad, so it just returns 00 to let the HCS-II know that the module is online. This status is indicated by an asterisk on the HCS-II host screen.

COMMAND PARSING

The PIC-TV uses the same serial routines outlined in my PIC-DIO article (*Circuit Cellar* 110). Even though the PIC-TV screen size is 308 characters, the buffer size is still 128 bytes because the HCS-II can't send more than 96 characters in a single network packet.

The code architecture is similar to the PIC-DIO. Packets are received and processed during serial interrupts and network packets are processed in a `main()` loop with `case` statements.

Because the PIC-TV's main function is to display text on a TV, most of the processing code revolves around handling string packets from the

Listing 1—Much of the PIC-TV code is dedicated to processing escape commands embedded in strings sent from the HCS-II. Functions such as module mode, cursor movement, and color commands are handled here. Some similar commands have been removed for brevity.

```
byte get_ansi_number() {
    int tval1, tval2;

    // Routine grabs numbers for ANSI commands out of current buffer
    // Used for both row & column numbers, which may be one or two digits
    tval1 = read_buffer(process_idx) & 0x0F;
    // Quick & dirty ASCII conversion
    tval2 = read_buffer(++process_idx);
    if ((tval2 > '/') && (tval2 < ':')) { // Add together and skip the ;
        tval1 *= 10; // Move first digit to tens position
        tval1 += (tval2 & 0x0F); // Grab ones digit & point to next char
        process_idx++;
    }
    return tval1;
}

>> The code below is called when a \e is found in a string
case 'e': // Escape char! Could be lots of things!
        // ESC[(idx);(scratch)(command letter)
        process_idx += 2; // Skip [ sign
        scratch = 0; // Default if not specified (idx is set below)
        idx = read_buffer(process_idx); // Figure out what digits we have
        if (idx > '/' && idx < ':') { // We have a number - let's get it
            idx = get_ansi_number();
            if (read_buffer(process_idx) == ';') { // We have a second number
                process_idx++; // Point to next number
                scratch = get_ansi_number();
            }
        } else {
            idx = 0; // Command with no number—set to default value
        }
    }
```

(continued)

HCS-II. All control commands are embedded inside a string command.

The PIC-TV understands about 35 commands that are sent inside an output-string command (S=). Table 2 lists the PIC-TV display command set. Compared to the BOB-II command set, you'll notice some commands are just reformatted and sent directly to the BOB-II.

However, many of the commands require more processing before being sent to the BOB-II. Many commands are preceded by an optional row/column pair, making the command parser a little involved.

Listing 1 outlines the routine used to parse escape commands (\e[]). As you can see, it is a big case statement with sections for each command. This technique means the code is easy to read and didn't waste much ROM space for the jump table.

The tricky part is pulling out the optional row and column numbers. They may or may not be there, and if they are, they can be one or two digits. If they aren't there, they must default to 0.

Once the numbers are scanned for and stored in the `idx` and `scratch` variables, the command letter is checked. Based on this command letter, snippets of code are called to perform the requested function.

`idx` and `scratch` may seem like odd variable names, but this code was originally done for an LCD interface project squeezed into a smaller PIC. RAM space was at a premium so I reused variables whenever possible.

Once I moved up to a bigger PIC with more RAM, I never switched to unique variable names. Why waste resources just because they're available?

CURSOR CONTROL

The BOB-II's cursor control command is limited to moving the cursor to a specific `x,y` location. This provides plenty of flexibility, but only if you know your position at all times. Tracking a cursor position in XPRESS would be next to impossible.

To use these ANSI cursor commands, the PIC-TV chip tracks the cursor position. When a cursor movement command is received, the cursor

Listing 1—continued.

```
// Execute routine based on command letter, Some will set this true
// to skip moving cursor
skipgoto = FALSE;
switch (read_buffer(process_idx)) {
  case 'A': // Cursor up
    if (y > idx) {
      y -= idx;
    } else {
      y = 0;
    }
    break;

  case 'B': // Cursor down
    y += idx;
    if (y > rows) { y = rows; }
    break;

  [other similar cursor movement commands]

  case 'f': // Move cursor to x,y
    y = idx; x = scratch;
    break;

  case 'g': // Beep for a given period of time
    if (idx > 10) idx = 10;
    beepout = 1;
    b_time = idx * 10;
    set_rtcc(0x00);
    timeok = TRUE;
    break;

  [other similar cursor movement commands]

  case 'K': // Clear to end of line
    for(idx = x; idx <= cols; idx++) {
      restart_wdt();
      putc(' ');
    }
    break;

  case 'J': // ANSI clear screen
    if (idx == 2) { // Clear Screen
      printf(send_uart_serial, "{A}");
      delay_ms(5);
      x = 0; y = 0;
    }
    skipgoto = TRUE;
    break;

  // Color commands
  case 'M': // Set screen color in local mode only
    if (local_mode && (idx < 8)) {
      printf(send_uart_serial, "{F%u", idx);
    }
    skipgoto = TRUE; // No need to move the cursor
    break;

  [Other similar color set commands]

  case 's': // Save current cursor position
    sx = x; sy = y;
    skipgoto = TRUE;
    break;

  case 'u': // Restore saved cursor position
    x = sx; y = sy;
    break;

  default:
    skipgoto = TRUE;
    break;
}
if (!skipgoto) { move_cursor(x, y); } // A few commands don't
// require cursor to move
break;
```

Table 2—The PIC-TV command set is based on the original LCD-Link command set. Commands are sent embedded in strings from an HCS-II XPRESS program.

Command	Description	Command	Description
\b	Turn display on	\e[#C	Move cursor right # columns
\c	Turn display off	\e[#D	Move cursor left # columns
\e	Escape character for cursor/color commands	\e[H	Home cursor to 0,0 (## optional)
\f	Clear screen	\e[#;#f	Move cursor to row,col #;#
\g	Sound beeper	\e[#j	Move cursor to row # (;# col optional)
\h	Enable character backgrounds	\e[s	Save current cursor position
\i	Disable character backgrounds	\e[u	Restore saved cursor position
\j	Switch to local video mode	\e[2J	Clear screen
\k	Switch to Genlock/Overlay video mode	\e[K	Clear line to end of row
\l	Color entire character grid with background color	\e[7h	Set wrap mode. Lines wrap to beginning of same line at end of row.
\m	Color only character cell with background	\e[7l	Set CR/LF mode. Lines wrap to beginning of next line at end of row.
\n	New line	\e[#g	Beep for # seconds (0–10)
\o	Blink on	\e[#M	Set screen color (0–7)
\p	Blink off	\e[#N	Set character color (0–7)
\r	Carriage return	\e[#O	Set character cell background color (0–7)
\t	Tab (4, 8, 12, 16, 20, etc.)	\e[#P	Set character outline color (0–7)
\x##	Output special character using byte code		
\e[#A	Move cursor up # rows		
\e[#B	Move cursor down # rows		

variables are altered and the BOB-II is sent the cursor's new x,y position.

This way, the PIC-TV can move the cursor using relative (up x, down y) or absolute values (go to row y). Another feature is the ability to save and restore the current cursor position.

Some commands check the boundary limits, while others do not. This was done from a usefulness versus extra-cycles-needed viewpoint.

With the cursor commands using relative movements, it made sense to check the boundaries because you wouldn't always know exactly where you are in your XPRESS code.

However, with the absolute position commands, it didn't seem worth the cycles needed to check the boundaries. The BOB-II ignores cursor positions that are out of bounds. When coding your XPRESS program, knowing that the limits are 11,27 is sufficient.

COLOR MADE EASY AND MORE

Changing character or background colors is simple. The PIC-TV grabs the color number from the HCS-II command and converts it to a BOB-II color command, which is sent out via the hardware UART. Reformatting color commands is simple, but using them can be a bit more involved.

Color can only be used when the PIC-TV is in local mode. When you set the screen color, the entire screen changes color. However, you can use a background color to paint the entire character grid. When you do this, you

can create a two-color screen with the character grid set to the cell background color and the border set to the original screen color.

In two-color mode, you can still change the character cell background color by using the outline color command. The background color will remain the same for all characters.

Photo 2 shows a screen display using internally generated color backgrounds.

You can change the color of the characters and character outlines. However, the characters are somewhat pale compared to the background.

During development, I used a cheap modulator and TV for testing. The characters all looked white no matter



Photo 2—The PIC-TV can generate detailed color backgrounds at the screen, grid, or character cell level. This allows it to draw attention to urgent information.

what color I set them to. I spent hours trying to find the “bug” that was breaking the color commands.

Turns out, it wasn’t the code after all. When I used my 32” Sony TV and fed the video into it or my DISH receiver, the character colors were much more defined.

Some combinations of character and outline colors cause instability in the display or make the characters look jagged. But who wants a blue character with a magenta outline anyway? Leaving the outline black gives the best character definition.

One feature of the BOB-II is the ability to change character background colors on-the-fly. If you set the PIC-TV to only color the character cell background (\m), you can have different characters with different color backgrounds.

One trick is to use cell backgrounds to display different color blocks. Send a space with a specific color background and you can display status levels with a tiny bargraph as in Photo 2.

One of the nicest features of the BOB-II is its ability to overlay monochrome text onto an existing video signal (see Photo 3).

But why stop there? When the motion detectors outside detect movement at night, the HCS-II turns on the IR illuminators and the video camera. Using an MCIR-Link, your TV jumps to a dedicated channel with the feed from the camera and PIC-TV.

Chimes alert you if you’re not sitting in front of the TV. The HCS-II then displays all relevant security and alarm info on the screen and the video camera displays a picture of...your dog.

OK, that’s a bit extreme, but how about having the PIC-TV let you

know when a car comes in the driveway? You can easily insert the PIC-TV between your cable box and TV. Because the PIC-TV display can be turned on and off by the HCS-II, you can decide when text is displayed on top of your favorite sitcom.

When the HCS-II senses a car in the driveway, it flashes a message in the corner of the screen. Pressing a specific button on the remote switches the TV to the security camera.

If you connect a caller-ID modem to your HCS-II, you can use the PIC-TV to show the caller’s number when the phone rings.

LOOKING AHEAD

Most of the I/O in the PIC-TV is serial so there were many unused I/O pins. With an eye towards the future, I brought them out to a 14-pin header.

The PIC-TV only uses about 45% of the ’16C63A ROM, so there’s plenty of room to add code for whatever add-ons anyone comes up with. Swap the ’16C63A for a ’16C73A and it could read analog values. Add more code to the 16C63A and allow 8 bits on the expansion header to be accessible like a DIO-Link port.

Using an off-the-shelf OSD module saved me development time and aggravation, and enabled me to concentrate on features instead of core operations of the character display.

Although it increased the cost, I think it made for a more stable design. And because the HCS-II network protocol is so straightforward, the PIC-TV can be used in just about any system with an RS-485 network.



Photo 3—The BOB-II can generate its own backgrounds if no external video signal is used. All text is monochrome when overlaid on an existing video signal. An ideal application of this mode is to use a security camera image as a background.

Next time, I’ll cover how to use the PIC-TV in your HCS-II system in more detail. The PIC-TV can do some neat tricks, so stay tuned.... ☒

Mike Baptiste graduated from Rensselaer in 1992 and currently 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.

REFERENCES

- HCS-II LCD-Link Manual, ftp.circuitcellar.com/CCINK/1992/Issue_27/LCDLink.zip
- HCS-II, www.cc-concepts.com/products/hcs/
- BOB-II Technical Data, www.decadenet.com/bob2/bob2.html
- Microchip PIC16C63A datasheet, www.microchip.com/10/Lit/PICmicro/16C6X/30605/index.htm
- 78M05 datasheet, www.national.com/ds/LM/LM341.pdf
- STV5730A datasheet, us.st.com/stonline/books/pdf/docs/4434.pdf

SOURCES

PIC16C63A

Microchip
(888) 628-6247
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

BOB-II

Decade Engineering
(503) 743-3194
Fax: (503) 743-2095
www.decadenet.com

PIC-TV chips and kits, BOB-II

Creative Control Concepts
(919) 304-3107
Fax: (919) 304-3107
www.cc-concepts.com

PIC C Compiler

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

FEATURE ARTICLE

Stuart Ball

What's the Count?

Build an AVR-Controlled Frequency Counter

If you've ever needed a simple, inexpensive frequency counter, the Count-4 may be just what you're looking for. Pay attention as Stuart recounts building this microprocessor-controlled autoranging frequency counter.



Do you ever need a simple, inexpensive frequency counter? If so, you can build the Count-4—a microprocessor-controlled, autoranging frequency counter that can operate to 60 MHz.

The frequency counter in Figure 1 shows the basic concept starting with a string of cascaded counters that count cycles from the input you're trying to measure. This count is captured at regular intervals (1 Hz, 0.1 Hz, etc.) in a set of latches.

The counters are simultaneously reset at the end of the sample interval, after the count is captured. The captured count is displayed on an LCD or an LED display. The display shows the number of counts accumulated in each sample interval, a direct measurement of frequency.

Suppose we measure a 5-MHz signal using a 0.01-s sample interval. The counters, starting at zero, will accumulate 50,000 counts in 0.01 s. At the end of the 0.01-s interval, the count will be captured in latches and the counters will be reset for the next sample interval.

Early frequency counters needed a lot of counter digits to get both accuracy and range. In the example just described, the frequency counter would display 5.0000 MHz. If we

wanted more accuracy than five digits, we'd need more counters.

And there is a tradeoff between accuracy and time. For instance, if we wanted to measure a 5-kHz signal with five-digit accuracy, it would take 10 s to accumulate that many counts.

In addition to the counters that measure the input frequency, a frequency counter needs a crystal-controlled timebase that must be divided down to get the reference clock. If we start with a 1-MHz timebase, we need six decade counters to get a reference frequency of 1 Hz. The accuracy of the measurement is directly related to the accuracy of the timebase crystal.

Finally, a frequency counter has to display the result, which means running the captured count (in BCD) through a display decoder circuit to drive a seven-segment display.

As a result, a five-digit frequency counter built with TTL logic typically uses more than 20 ICs. Modern frequency counters use the same techniques but use a microprocessor to read, convert, and display the count, which reduces the number of components and provides more flexibility.

I developed the Count-4 because I needed a simple frequency counter. I rarely need more than four digits of accuracy, so the Count-4 uses a four-digit LED display. Three additional LEDs indicate whether the measured frequency is in hertz, kilohertz, or megahertz. Also, I didn't want switches for selecting the sampling timebase, so the Count-4 is autoranging.

HOW THE CIRCUIT WORKS

Figure 2 shows the heart of the Count-4 to be an Atmel AVR 90S4414 microcontroller (U1). The '90S4414 manipulates the counter ICs, performs autoranging functions, and displays the result on the LED display. For more information on AVR parts, take a look at the Atmel AVR Series sidebar on page 38.

The '90S4414 is connected to four 74ACT161 counters (U2–U5). The 74ACT161 is a four-bit synchronous binary counter, capable of operation to around 100 MHz. The 74ACT161 inputs include count enables, which synchronize multiple counters.

The counters count up when both enables (pins 7 and 10) are high, and stop counting when either enable is low. The enable input to the least significant counter (U5) is connected to the OC1B output from the '90S4414.

The '90S4414 can read the counter contents on ports A and C, and can reset the counters. Most importantly, the OC1B output of the '90S4414 can be programmed to toggle every time an internal counter rolls over.

To measure a frequency, the '90S4414 resets the counters and forces the OC1B output low. Then the OC1B output is driven high for 1 ms and the count is read. If the value in the counter is greater than 4095 (1000 hex), the count is assumed to be scaled properly.

If the count is less than 4095, the process is repeated for 0.01-, 0.1-, and 1-s count periods. Because 1 s is the longest gate time, whatever count is captured will then be displayed, even if it is zero. Starting with the shortest gate time permits autoranging because the processor always finds the gate interval that provides the proper scaling.

The 16-bit counter can count up to FFFF hex, or 65535 decimal. This number corresponds to 65.5 MHz with a 1-ms gate time, and determines the upper frequency limit.

After the count is captured, it must be displayed. The 74ACT161 counters are binary, but we want our display to be in decimal, so the firmware converts the 16-bit binary to five BCD digits.

After conversion to BCD, the most significant digit of the count is examined. If it is nonzero, then the upper four digits of the five-digit count are displayed. If the high digit is zero, then the lowest four digits are displayed. The correct position for the decimal point is also calculated.

Capture for 1 ms
 Result = 64 hex (100 decimal)
 Result < 1000 (hex), so capture for 0.01 s
 Result = 3E8 hex (1000 decimal)
 Result < 1000 (hex), so capture for 0.1 s
 Result = 2710 hex (10,000 decimal)
 Result > 1000 hex, so convert to BCD
 BCD result = 10000
 MS digit is nonzero, so display 100.0 and turn on kHz LED

The LED display is also driven by the '90S4414. The display is a four-digit module with common anodes, which means the cathodes of the same segment on each display digit are connected together. So, all the "a" segment cathodes are common, as are all the "b" segment cathodes, and so on.

Although the cathodes are common across the digits, the anodes for each digit are tied together. So, to display a "0" in digit 1, all the cathodes are driven low except the "g" segment, and the anode of digit 1 is driven high. Because digit 1 is the only digit with current supplied to the anode, the other three digits remain dark.

All four digits are cycled on and off one at a time and rapidly enough to appear as a continuous display to the eye. A 200-Hz interrupt causes each digit of the display to be activated for 5 ms, giving a total display scan time of 20 ms. No display driver IC is needed to convert the BCD count to seven-segment format, because this conversion is done in firmware.

The Port B outputs of the '90S4414 are capable of sinking 20 mA, so they can directly drive the LED cathodes (with 330-Ω current-limiting resistors). The display anodes are driven to +5 V using 2N4403 PNP transistors to individually enable each digit.

The transistors are needed because the current into the anodes is the sum of the individual segment currents, and can reach 90 mA. This current is well beyond the source current available from the '90S4414 outputs.

The frequency range (hertz, kilohertz, or megahertz) is displayed by three individual LEDs. These three LEDs use a common current-limiting resistor because only one LED at a time will be turned on. A fourth LED indicates when the gate (enable) signal is active.

The input amplifier for a frequency counter is tricky. It has to take a low-

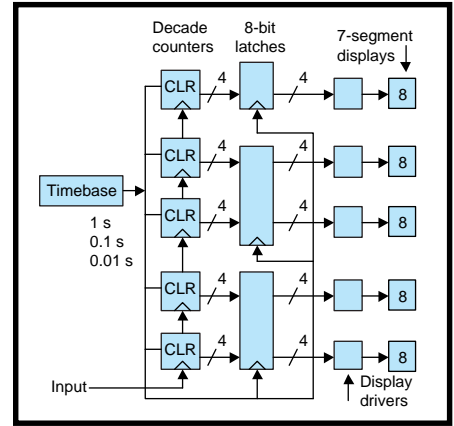


Figure 1—Here is a block diagram of a typical frequency counter implemented with discrete logic. A circuit to implement this may have more than 20 ICs.

level input signal to logic levels, and do it over a frequency range that goes from hertz to tens of megahertz.

The input amplifier for the Count-4 consists of a feedback-biased 2N2222 transistor (Q5). The 1-mH inductor in the collector circuit of Q5 provides additional gain at higher frequencies, where the transistor gain may fall off. The circuit works past the 60-MHz limit of the Count-4.

Power to the counter is supplied by an external 9-VDC transformer and regulated with a 7805 (U6) to produce the 5 V needed by the logic.

THE FIRMWARE

The 90S4414 firmware is written in assembler, using the Atmel cross-assembler, available on their web site. The '90S4414 has two internal counters, T0 and T1.

Timer T1 generates the measurement timebase. To generate a 1-s gate time, the 4.096 crystal must be divided by 4,096,000, which is beyond the range of the 16-bit T1 timer.

The reason a 4.096-MHz crystal was chosen is because the '90S4414 timers include a prescaler that can divide the crystal frequency by 8, 64, 256, or 1024. Starting with 4.096-MHz, these divisors will all result in an intermediate frequency that can be divided by an integer to get the needed gate frequencies.

To generate the gate signal, the prescaler is programmed to divide the crystal frequency by 256, producing a 16,000-Hz intermediate frequency. This frequency is then divided by

Table 1—Scaling is accomplished by capturing for increasing time periods until the count rises to a level high enough for accurate display. This example captures 1 ms, 0.01 s, and 0.1 s before the count is large enough to display.

timer T1 to produce the 0.001-, 0.01-, 0.1-, and 1-s gate intervals on OC1B.

A measurement cycle starts by resetting the counter (by toggling Port D, bit 7) and resetting the OC1B outputs. The OC1B output can either toggle, set, or reset when T1 rolls over.

To guarantee that OC1B starts in the right state, the timer is programmed to reset OC1B and a short timing cycle is run. Then the timer is programmed to toggle OC1B and set to roll over at 16 counts, producing a 1-ms toggle rate. The timer is allowed to count for two rollovers.

The first T1 rollover takes 1 ms and leaves the OC1B output in the high state, enabling the counters. The second rollover toggles OC1B low, stopping the counters without resetting them. T1 is then stopped and the count is read on ports A and C, converted to BCD, and sent to the display locations.

If the count is too small to be displayed, the sequence is repeated with longer gate times up to 1 s. Because a new count is acquired as soon as the previous count is captured and displayed, higher frequencies will have a

faster update rate than slower frequencies.

To illustrate how autoranging works, the sequence for measuring a 100-kHz signal is shown in Table 1. Although the T1 timer can generate an interrupt on rollover, the firmware isn't doing anything else when waiting for the rollover to occur, so it polls the timer for the rollover condition.

The firmware continuously executes a loop, collecting and displaying frequencies. Common subroutines clear the counter, acquire a new count, and convert the result to BCD.

Timer T0 is an 8-bit counter and generates the 200-Hz LED refresh clock. Again, an internal prescaler divides the 4.096-MHz crystal frequency by 256 to get 16,000 Hz. T0 then divides this by 80 for a 200-Hz interrupt.

The LED display interrupt service routine (ISR) turns off all the displays, selects the next display digit, converts the BCD digit value to a seven-segment value, and writes the result to Port B. The decimal point is ANDed into the value on Port B if the current

Atmel AVR Microcontrollers

The Atmel AVR-series of microcontrollers includes the 20-pin AT90S1200 and AT90S2313, and the 40-pin AT90S4414 and AT90S8515 devices. The AVR processors are 8-bit RISC-based machines, and all execute the same instruction set (except the '2313, which executes a subset of those instructions).

AVR processors execute about one instruction per clock cycle, which lets the processor run at slower clock rates (for a given level of performance) than processors that use a higher frequency clock and divide it internally to produce the instruction clock. The Microchip PIC processors, for example, require a 20-MHz input (internally divided by 4) to achieve the same instruction execution time as a 5-MHz AVR processor.

The AT90S4414 used in the Count-4 project includes 4 KB of flash program memory, 32 registers, 256 bytes of SRAM, 256 bytes of EEPROM, and 32 I/O lines. There are two flexible timers, one 8-bit and one 16-bit, that support several modes of operation.

The most important feature of the device for this project is the ability to toggle an output pin when a timer rolls over. Other features include synchronous and asynchronous serial interfaces, an analog comparator, a watchdog timer, and in-circuit programming capability.

The 32 general-purpose registers are divided into two groups of 16 registers. One group, registers 16–32, can execute immediate instructions such as Load Immediate, Compare Immediate, and so on. Other than this, all the registers are exactly the same, making the instruction set quite flexible. Some of the registers have dual uses, such as pointers for table lookup instructions.

digit requires a decimal point to display. Then the appropriate transistor is turned on to enable that display digit. Last, the correct band LED is turned on.

Although the AT90S4414 includes 256 bytes of SRAM, the counter is implemented entirely using the 32 general-purpose registers. Which band LED (hertz, kilohertz, megahertz) to turn on is determined when the count is examined and the decimal point position is calculated.

For instance, the algorithm for the 1-s gate interval looks like this: if the upper digit (fifth position) is nonzero (meaning the count is >10,000 Hz), then display the upper five digits, turn on the kilohertz LED, and place the decimal point after the LED hundreds digit. If the upper digit (fifth position) is zero (count is <10,000), then display the lower four digits, turn on the hertz LED, and so on.

CONSTRUCTING THE CIRCUIT

The prototype was constructed on perfboard using point-to-point wiring and was mounted in a plastic Radio Shack enclosure. A hole was cut in the enclosure and covered with smoked plastic to provide a viewing window for the display and LEDs.

Be sure to include the bypass capacitors, one near the '90S4414 and the rest near the counters. Because

the counters will be operating at high frequencies, I recommend that they be grounded by running strips of adhesive-backed copper tape under the ICs and along the edges of the perfboard to provide a fairly low-impedance ground path. This is my normal method of making ground connections in any digital prototype. It isn't quite as good as a ground plane, but it's close.

The LED display is a four-digit, green, common-anode display (Lumex LDQ-M282RI). Any common-anode, seven-segment LED digits and wire all the cathodes in parallel to make a four-digit display if you want. Of course, if you use a different display, it will probably have a different pinout from that seen in Figure 2.

The prototype was constructed with the LEDs and four-digit display on the back of the board. This will provide access to the components once the board is mounted in the case.

Power on the prototype comes from a 2.1-mm coaxial power connector, but you should use whatever matches your 9-VDC power transformer. If you have a power supply that puts out regulated 5 VDC, you won't need the '7805 regulator (U6).

Because the circuit has to operate at fairly high frequencies, the wiring of the counter clocks (pin 2) needs to be fairly short, and the input amplifier wiring needs to be very short. The Gate LED (D4) is optional.

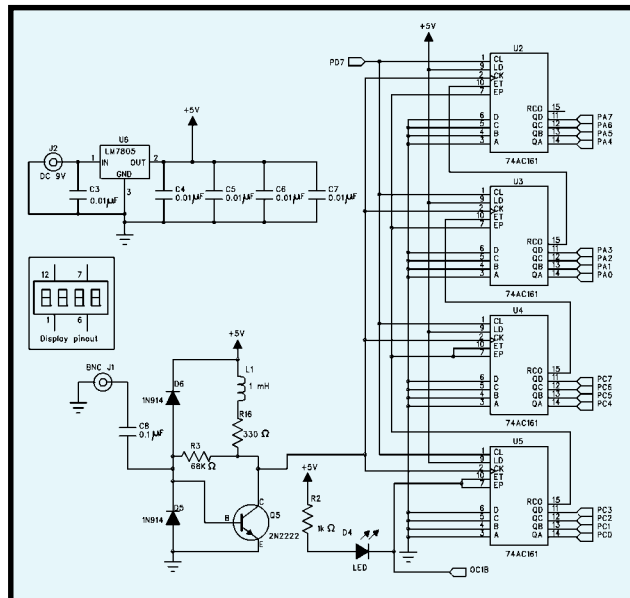


Figure 2—The counter outputs are read by the microprocessor at the end of each sample interval.

I built a short (6") input cable with alligator clip leads that let me connect an antenna or a pickup coil or connect the counter input directly to the circuit I want to test.

CHECKOUT

First, check the wiring of the +5-V circuit. Before installing the ICs, plug in the 9-V transformer and check for 5 V at the IC pins. If the voltage is wrong, check the wiring of the '7805 regulator.

After verifying that the power supply is working, install the ICs and plug in the transformer again. You should see the LED display 0000, and the hertz LED should be illuminated. If you installed the gate LED, it should be blinking.

Finally, apply an input signal to the counter and verify that it displays the correct frequency. I built the prototype with an extra socket that lets me plug in a TTL oscillator to verify that everything works (see Photo 1).

The Count-4 is easy to use because there are no knobs, buttons, or switches. Just hook it up and look at the resulting display (see Photo 2).

The input amplifier is designed to produce a logic-level output from fairly low (50 mV) inputs, so it's possible to saturate it. If you need to measure high-amplitude (>1 V) signals at frequencies over about 100 kHz,

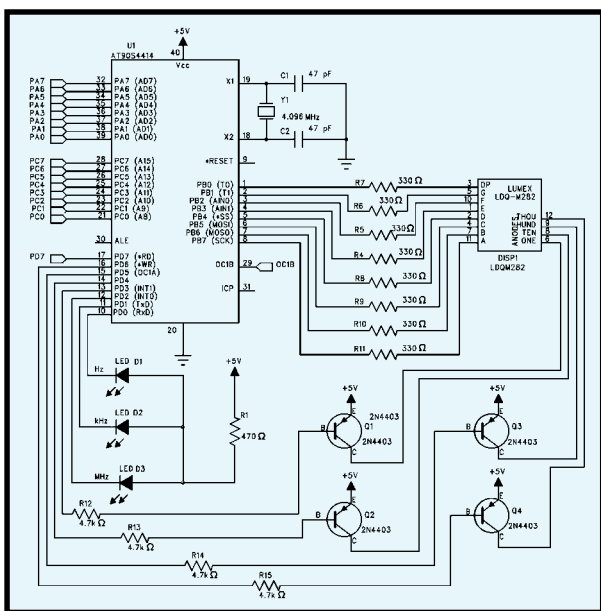


Figure 3—The microprocessor reads the counter outputs and drives the 4-digit, 7-segment display. Display anodes are driven by transistors Q1–Q4. The OC1B output of the microprocessor enables the counters.

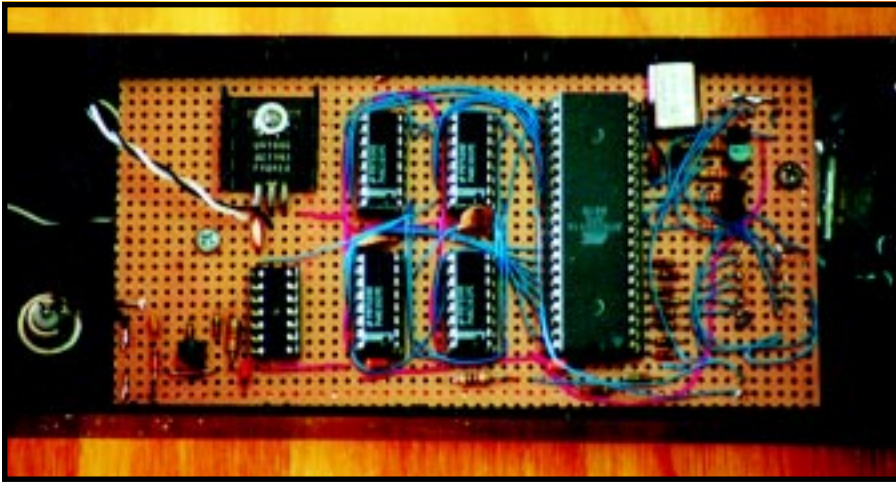


Photo 1—Here's a look at the interior of the Count-4. The LED display and the band LEDs mount on the back of the board. The empty 14-pin socket is for a DIP-style crystal oscillator, used to test and debug the circuit.

put a 1-k Ω resistor in series with the counter input. This will also limit the loading effect of the input amplifier on whatever you're measuring.

As with any piece of test equipment, it's possible to apply a signal that will damage the input amplifier. Unless you're measuring the frequency of the high-voltage stage in a television or trying to directly measure the output of an amateur transmitter, this is not likely to be a problem. Protection diodes (D5 and D6) may distort the measured waveform under some conditions, though.

My experience with frequency counters, even expensive ones, is that the input circuit tends to affect the frequency of whatever you're measuring when you connect to RF circuits. I usually use a pickup coil to measure RF, to help reduce this effect. If the

signal is too strong and overloads the input amplifier, move the pickup coil away. If the measured signal is stable and the display is unstable, you may be overloading the input stage.

GOING FURTHER

The Count-4 isn't designed for any frequency higher than 60 MHz, but the upper limit can be extended. The primary limitation on the upper frequency of the Count-4 is the 16-bit counter; with a 1-ms gate, the maximum count corresponds to 65 MHz.

Unfortunately, the next logical step up to a 100- μ s gate is not possible using a 4.096-MHz crystal. The divisor would be 409.6, which is not an integer.

Frequencies up to about 100 MHz could be measured by using a 500- μ s gate and doubling the result (in firmware) prior to the BCD conversion. You'll need to shift the binary count left one position, with an additional register to hold the potential overflow. This essentially creates a 17-bit result and you can discard the lower four bits.

Once you get past the limit of the 16-bit counter, you'll reach another limit, which is the maximum count frequency of the 74AC161 counters



Photo 2—There are no knobs or switches on the Count-4. Just plug in the power supply and hook up the input.

(around 100 MHz). You can improve this by adding a Schmitt-trigger buffer (such as a 74AC14) between the input amplifier and the counters to reduce the capacitive loading on the amplifier output.

The upper frequency limit can be further extended by using an ECL divide-by-10 prescaler between the amplifier and the counters. Devices such as the Motorola MC10E136 can operate up to 500 MHz, with the proper input amplifier design. The output has to be converted to CMOS levels and the resulting display will be one-tenth of the actual frequency.

Finally, the Count-4 could be modified to display more than four digits. Four more digits can be driven by using two four-digit displays and controlling the anode drive transistors through a 74AC138 decoder.

By far the simplest way, though, is to switch to an LCD, which would let you do away with the band LEDs, displaying that information on the LCD instead. In either case, you have to change the firmware to match.

One word of caution: if you attempt to increase accuracy by adding more digits, you will eventually run into the tolerance limitations of the 4.096-MHz crystal and the propagation delays inside the AT90S4414. There is a practical limit to the measurement accuracy with this circuit.

And that's all there is to it. As you can see, the Count-4 is easy to build and provides a simple, direct way to measure frequency. 📧

Stuart Ball works at Organon Teknika, a manufacturer of medical instruments. He has been a design engineer for 19 years, working on projects as diverse as GPS and single-chip microcontroller designs. He has also written two books on embedded-system design. You may reach him at sball85964@aol.com.

SOFTWARE

The software for the Count-4 may be downloaded via the *Circuit Cellar* web site.

SOURCES

AVR90S4414

Atmel Corp.
(408) 441-0311
Fax: (408) 436-4200
www.atmel.com

Marshall Industries
(800) 833-9910
(626) 307-6000
Fax: (626) 307-6173
www.marshall.com

Arrow CMS Distribution Group
(888) 263-7720
www.arrow.com

Insight Enterprises, Inc.
(800) INSIGHT
(480) 902-1001
Fax: (480) 902-1180
www.insight.com

LDQ-M282RI

Lumex, Inc.
(847) 359-2790
Fax: (847) 359-8904
www.lumex.com

PC/104-BASED WINDOWS CE DEVELOPMENT KIT

The **Élan-104 development kit** for Windows CE combines the Arcom Élan-104 processor board with Microsoft's Windows CE 2.11 to offer true fast-track applications development capability. The Élan-104 is a high-performance, PC/104-compatible embedded PC that features a 100-MHz AMD Élan SC400 '486SX microcontroller with 4-, 8-, or 16-MB EDO DRAM and a 4 or 8-MB flash memory preinstalled with Datalight ROM-DOS and Flash FX. The compact, single-height Eurocard format (100 × 160 mm) board includes full power management and all standard PC peripherals, including flat-panel interface.

The development kit includes the Élan-104 with 16-MB DRAM, 8-MB flash memory, and a preconfigured build of Windows CE 2.11 (tailored specifically for the Élan-104) preloaded into an onboard flash-based drive. Also included are a 5-V power supply, mouse, serial, and floppy disk/VGA interface cables. An optional 6.5", 640 × 480 color flat-panel display is available. Software in the kit includes the Windows CE Platform SDK for Élan-104, sample code, and demo applications, full documentation, and backup copies of the Windows CE image on a floppy disk.

The ÉLAN-104 Windows CE development kit sells for **\$995**.



Arcom Control Systems
(888) 941-2224 • (816) 941-7025
Fax: (816) 941-7807
www.arcomcontrols.com

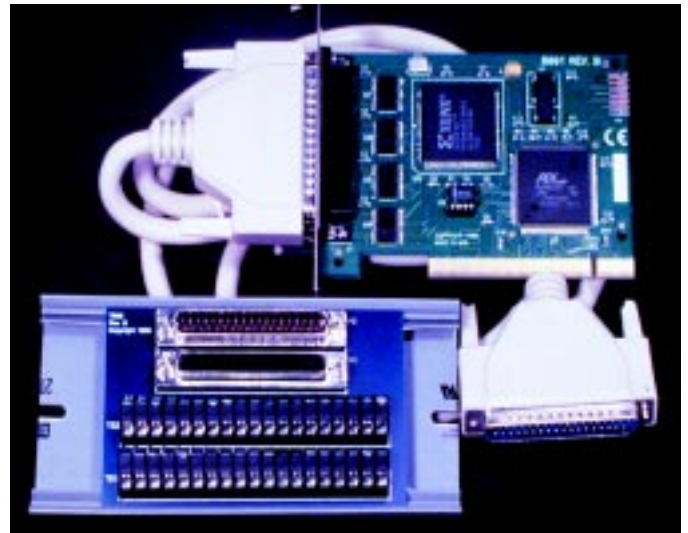
32-CHANNEL PCI DIGITAL I/O CARD

The **PIO-32.PCI** uses four I/O ports to provide 32 channels of configurable digital I/O. Port status is user-selectable as input or output by writing a control word to the port register. Any of the ports can be set to generate an interrupt for status monitoring. Applications include PC-based control and automation of equipment such as sensors, switches, satellite antenna control systems, video and audio studio automation, and security control systems.

Included with the card is the Seal/O suite of Windows 95/98/NT drivers. Seal/O provides a straightforward API. A utility for configuring the driver parameters under Windows 95/98 and Windows NT is also included. Popular development environments, including Visual C++, Visual Basic, and Delphi, support application development.

Seal/O TST, a Windows 95/98/NT console application is included to enable the user to exercise the inputs and relays. Source code is included to aid software development. Seal/O VB, a 32-bit Visual Basic sample with GUI allows control of individual or groups of relays, timed relay activation, and input monitoring.

An optional terminal block kit, **KT-101**, can be used to interface to the card's DC-37 connector. The kit consists of a 6" male/female cable and positive tension screw terminal block to provide a simple means to connect field wiring.



The PIO-32.PCI sells for **\$199**. The KT-101 is priced at **\$49**.

Sealevel Systems, Inc.
(864) 843-4343 • Fax: (864) 843-3067
www.sealevel.com

Nouveau PC

edited by Harv Weiner

COMPACT WEB-SERVER TCP/IP CONTROLLER

The **μFlashTCP** is a low-cost web-server TCP/IP controller in a package slightly larger than a credit card. It features a 25-MHz Intel '386EX processor with 512 KB of SRAM, 512 KB of flash memory, a watchdog timer, and 10 DIO lines. Two PC-compatible serial RS-232 ports are provided, along with the ability to configure one of the ports as RS-485. Ethernet support is provided by an NE-2000-compatible controller with 16 KB of on-chip buffer memory and 10BaseT interface. The μFlashTCP also supports M-Systems' DiskOnChip, providing nonvolatile mass storage from 2 to 144 MB.

The μFlashTCP includes pre-installed DOS, utilities, and web-server software. WATTCP TCP/IP stack libraries are integrated into the Borland C/C++ development environment. Source code for the

web server along with other TCP/IP services is included in the development kit. Also available is

a toolkit that adds real-time multitasking extensions and many examples to the Borland C/C++ development environment.

The μFlashTCP sells for **\$169** in quantities of 100. The μFlashTCP development kit is priced at **\$399**, and the TCP/IP multitasking software toolkit is **\$799**. An inexpensive I/O expansion board with ADC, DAC, relay drivers, and protected inputs is also available.



JK microsystems, Inc.
(530) 297-6073
Fax: (530) 297-6074
www.jkmicro.com

Nouveau PC

Serial Port Interfacing

Having worked on plenty of projects that use this common interface, sure, Ingo can give us the full scoop on serial port interfacing in general. But he'll also focus on the software required to talk to the serial port under (surprise!) Linux.

Serial ports are probably the most commonly found interface on computers in general, and specifically on PCs. In the past, I've written about projects that interface with serial ports—for example, the series on the global positioning system (GPS) that I completed last month.

GPS receivers use serial ports to communicate with the host. Some of you have written to me that you'd like to know how to write software to talk to serial ports, in particular under Linux. So here we go....

The PC serial port has been part of the system since early on. At first, the serial port was an add-on card based on the Intel 8250 UART chip. This serial interface card had a programmable data-rate generator, which, for its time, was pretty advanced. Many serial interface cards for other microcomputer systems had to be programmed via jumpers.

Although serial ports are now usually included on the motherboard, and in some cases even on the chip itself (e.g.,

the Intel '386EX or other embedded processors), the programmer's model hasn't changed much. In fact, if you really wanted to, you could plug an original IBM serial adapter card into your new P2 600-MHz computer. Well, you might have to adjust the AT bus clock down to 5 MHz, since these old cards can't run at 8 MHz.

However, the software, Windows, and most embedded operating systems would still work with this card. This compatibility is what makes serial ports a popular choice when it comes to choosing a standard interface.

The downside, at least as perceived, is that serial ports are slow. But this of course depends on the application. Most PCs can run at 115 kbps with newer UARTs, while

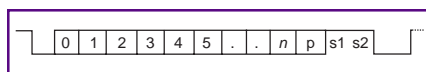


Figure 1—In the basic structure of an asynchronous serial word, the start bit is followed by 5–8 data bits, an optional parity bit, and 1–2 stop bits.

some can even run at 230 kbps. As we'll see later, that equates to about 11.5 and 23 Kbps. You can't run CD-quality digital audio without compression, but telephone-quality sound is possible.

PC SERIAL HARDWARE

As I mentioned, the UART used on PCs is based on the i8250. Several variants of this chip are now in use. These are the 16x50 series UARTs—the 16450, 16550, and 16650. They differ mostly by their performance and can be thought of as an extension of the basic i8250.

The UART's primary function is to convert parallel data to serial in the transmitter and convert received serial data to parallel in the receiver. The serial data rate is determined by the bit-rate clock, which is derived from a crystal oscillator with a programmable prescaler. In the PC, this clock is based on a 1.8432-MHz crystal.

Internally, the receiver and transmitter module is clocked at 16 times the desired

bit rate. So a 1.8432-MHz crystal gives a maximum data rate of 115.2 kbps, when the divisor of the prescaler is set to 1.

The transmitter and receiver are also programmable to the number of data bits in the serial word and the number of stop bits, as well as type of parity used. A generic serial word is shown in Figure 1.

As you see, it begins with a start bit. This low bit on the serial line indicates the start of a word. Next, we have 5–8 data bits, depending on how the UART is programmed, and following the data bits is the optional parity bit. The parity bit can be missing (none), set to one (space), set to zero (mark), or even/odd parity.

Finally, there are 1–2 stop bits. But, stop bits aren't really bits; they're just a guaranteed guard space between serial words to make sure that there will be a high-to-low transition at the start of the word.

COM1	0x3f8
COM2	0x2f8
COM3	0x3e8
COM4	0x2f8

Table 1—This table lists the standard base address for UARTs on a PC. Each UART takes 8 bytes of I/O space, starting at the base address.

On the receiving end, things are more automatic. The receiver just waits for a start bit and deserializes the data into the receive buffer. Once that's done, it asserts a flag (receive buffer full) and interrupts. The host software then reads the character from the receive buffer register (RBR).

Receive errors occur when the parity does not match (parity error) or when there aren't any stop bits at the end of the word (framing error). If a new serial word is decoded before the CPU had a chance to read the old one, we can get an overrun error.

Overrun errors were common at high data rates in the earlier UARTs and prompted the design of UARTs with larger receive buffers (16550, 16650, etc.).

In addition to the basic functions of the UART, the i8250 also handles some of the modem control signals—DTR, DSR, RTS, CTS, RI, and DCD. These signal the state of the serial connection, especially when used with a modem.

Let's look at the registers in the 8250. The 8250 uses 8 bytes of I/O space. Typically, the UART is at one of the addresses in Table 1.

The registers can be read- or write-only as well as read and write. Some registers are overlapped. The register layout is shown in Table 2.

I mentioned the RBR and THR already. A bit in the interrupt enable register (IER) tells the UART which events on the UART can cause an interrupt request. Possible sources are changes in the modem status signals (DSR, RI, CTS, DCD), receive errors (parity, overrun, framing error, or break conditions), and transmitter buffer empty or receiver buffer full. Once an interrupt occurs, the software can check the interrupt identification register (IIR) to find out the cause.

The data framing register (DFR) is used to program the serial data word format (word size, stop bits, parity type). It can also be used to program a break condition on the transmit line.

One special bit in this register is the data latch address bit (DLAB), which programs the behavior of the lower two registers on all UARTs and the extended functions registers on 16650 on up.

Note that in Table 2, there are two columns in the first two locations (0x00, 0x01, 0x02). We can access the divisor register when the DLAB bit in the DFR is set to one; otherwise, the UART's data registers (THR, RBR) and the IER are accessible.

The modem control register (MCR) is a plain parallel port that lets you program the state of the modem control signals (DTR, RTS). It has two general-purpose output bits (OUT1, OUT2), which are not used for anything on a PC, and the loop-back bit.

The loop back lets you put the UART in internal loop-back mode. In this mode, all of the transmitted data is echoed back in to the receiver. This mode is useful for testing software and isolating hardware errors.

The line status register (LSR) contains transmitter empty and transmit holding buffer empty flags, the receiver data ready flag, and receiver error flags (break, framing, overrun, parity). The transmit holding buffer empty flag tells us when the UART is ready to consume the next character, and the transmitter empty flag tells us when the last character was sent out.

The modem status register (MSR) lets you monitor the state of the modem status lines (DSR, DCD, RI, CTS). It's a simple input register that also latches if there is a transition of any of the lines.

Finally, the scratchpad register (SPR) is just a general-purpose register that can store a byte of data. You can use it in your software to store flags or status information about this particular serial port. The SPR is only available in 16450 and newer UARTs—not the 8250.

That brings me to a quick summary of the 8250 variants that are used in PCs. Table 3 lists the differences. Starting with

Offset	DLAB = 0		DLAB = 1	
	Read	Write	Read	Write
0x00	RBR	THR	DLRI	DLRI
0x01	IER	IER	DLR _h	DLR _h
0x02	IIR	FCR	EFC	EFC
0x03	DFR	DFR	DFR	DFR
0x04	MCR	MCR	MCR	MCR
0x05	LSR	LSR	LSR	LSR
0x06	MSR	MSR	MSR	MSR
0x07	SPR	SPR	SPR	SPR

Table 2—In this register layout of a UART, the left column indicates the registers that are available with DLAB = 0, whereas the right column reflects the layout for DLAB = 1.

When the software wants to transmit data, it deposits a word into a transmit hold register (THR). The UART wakes up, serializes this data word, and adds the start/stop and possibly the parity bit. The UART consumes this character, sets a flag (transmit hold buffer empty), and interrupts the host.

In CPU time, transmitting a character takes forever. Consider that if we transmit the most common word format (8N1) at 9600 bps, we have to transmit 10 bits per word. This means it takes:

$$\text{time} = \frac{10 \text{ bits}}{9600 \text{ bps}} = 1.04\text{ms}$$

At 100 MIPS, that's 100,000 instructions.... Although you could sit in a loop waiting for the transmit-done flag, you really want to use interrupt-driven I/O for serial ports.

i8250	9600 bps	single char
ns16450	115,000 bps	single char
ns16550	115,000 bps	16-byte FIFO
ns16660	115,000 bps	20-byte FIFO
ns16760	230,000 bps	64-byte FIFO

Table 3—There are different types of UARTs available for the PC. If you plan on doing any kind of high-speed serial I/O, you should make sure to use a 16550 or better.

thens16550, the buffer size was increased from a single-byte to multiple-byte FIFOs.

On these UARTs, the FIFO control register (FCR) enables the receiver and transmitter FIFO and sets the receive FIFO threshold level. The threshold level is the number of bytes that must be received before the UART causes an interrupt. This requirement reduces the interrupt rate in the system, so the interrupt handler can read multiple bytes at each interrupt.

Ready to do some programming? Under most circumstances, you'll be running in an operating system environment and chances are that this OS already implements the device drivers necessary to let your application access serial devices. This is true under Windows, Linux, and many RTOSs. These drivers work well for general-purpose applications and are easy to use in your software.

Listing 1 shows how to open a serial device and start using it under Unix-like operating systems. This example opens a serial device, which is identified with a symbolic name (e.g., /dev/ttyS0), sets

Listing 1—This code opens and programs a serial port under Linux (and most Unix systems). The device name is usually something like /dev/tty00. You use the ioctl system call to set the data rate, word size, as well as the operating modes of the serial driver.

```
struct termio svbuf;
fd = open("/dev/ttyS0", O_RDWR | O_NDELAY);
ioctl(fd, TCGETA, &svbuf);
svbuf.c_iflag = 0;
svbuf.c_oflag = 0;
svbuf.c_lflag = 0;
svbuf.c_cflag = B38400 | CS8 | 2 | 0 | CLOCAL | CREAD;
ioctl(fd, TCSETA, &svbuf);
```

Listing 2—This listing handles I/O with a serial port under Linux. You use the read and write system call to access the serial port, once it has been opened. If you are operating in nonblocking mode, the read and write can return only partially satisfying the request.

```
/* basic send command to send string to SX Blitz */
sx_send(fd, cmd, len)
int fd;
unsigned char cmd[];
int len;
{
    int i, n, r, try;
    unsigned char c;

    for(i=0; i<len; i++){
        n = 1;
        try = 10;
        while(n != 0 && try != 0){
```

(continued)

Listing 2—continued

```
    r = write(fd,&cmd[i],1);
    if( r > 0){
        n -= r;
    }else{
        usleep(10);
        try--;
        fprintf(stderr, ".");
    }
}
if(!try){
    fprintf(stderr, "Timeout!\n");
    return(-1);
}
n = 1;
try = 10;
while(n != 0 && try != 0){
    r = read(fd,&c,1);
    if( r > 0){
        n -= r;
    }else{
        usleep(10);
        try--;
        fprintf(stderr, ".");
    }
}
if(!try){
    fprintf(stderr, "Timeout!\n");
    return(-1);
}
if(c != cmd[i]){
    fprintf(stderr, "Cmd/resp mismatch!
                %02x %02x\n", c, cmd[i]);
    return(-1);
}
}
return(0);
}
```

Listing 3—As you see in this RT-Linux interrupt mode serial driver example, there are no generic serial drivers under RT-Linux, since it's hard to write a general serial driver that will work for all real-time applications. This driver echoes characters that it receives.

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>

/*#include <linux/rt_sched.h> */
/*#include <linux/rtf.h> */
/*#include <asm/rt_irq.h> */

#include <linux/mc146818rtc.h>
#include <linux/serial_reg.h>

/*#include <linux/cons.h> */

char buf[16]; /* circular buffer */
int pi,po;

#define PORT(x) (0x3f8+x)
/* set these for your hardware */
#define IRQ 4
#define MOD_8N2 (UART_LCR_WLEN8|UART_LCR_STOP)

void intr_handler(void)
/* handle interrupts from UART */
```

(continued)

the transfer rate, the number of data bits, stop bits, and parity.

The serial device driver under Unix does many great things—line buffering, it implements line-based editing, and more. All the stuff you need to hook up a terminal to it.

However, in most embedded applications we don't need that. We just want to send and receive characters. Listing 1 tells the serial port driver to just act dumb and let the application handle it.

You may have noticed that I opened the device with the `O_NDELAY` flag, which tells the OS that I want to use nonblocking I/O. In nonblocking I/O, a read or write operation will not block when the read buffer is empty or the transmit buffer is full.

Once the device is open and programmed, we can start using it. This is done by using the read and write calls. Listing 2 shows a typical example.

The functions read and write take three arguments—a file descriptor, a pointer to a buffer, and the number of bytes requested in the operation. In this example, I implement the low-level protocol for a device programmer. The protocol is simple: the programmer echoes all the bytes sent to it.

Let's start by trying to send all of the bytes requested by the calling program. Here we send each byte with a write function and look at the return value. If the return value is greater than 0, the serial driver had enough buffer space to accept the byte. Since we are using nonblocking I/O, we poll if we're not successful in sending the byte.

Once the buffer is sent, we try to read the data back using the read function, which we also poll. Once all of the data is read back and compared to the transmitted data, we return. If we timeout (i.e., reach the "try" count before completing), then we return an error value (-1) to the caller to indicate that we couldn't send the command to the programmer.

My programmer is a SX Blitz from Parallax that will program an SX CPU from Scenix. You can download the complete program from the *Circuit Cellar* web site.

Like anything else, using the OS drivers for serial I/O has advantages and disadvantages. Of course, it's easy to use because the OS has abstracted the serial port into a nice file-oriented object. But the

Listing 3—continued

```

{
int irqsrc;
int msk;
msk = UART_IER_RDI;
while(1){
    irqsrc = inb(PORT(UART_IIR));
    if(irqsrc & UART_IIR_NO_INT) /* nothing pending */
        break;
    switch(irqsrc & 0x06){
        case UART_IIR_RDI:
            /* receive data interrupt loop while there are */
            /* chars to process */
            while(inb(PORT(UART_LSR))&UART_LSR_DR){
                if(((pi+1)%16) != po){
                    buf[pi] = inb(PORT(UART_RX));
                    pi = (pi++)%16;
                }else{ /* no room, pitch */
                    inb(PORT(UART_RX));
                }
            }
            /* fall through to xmit */
            case UART_IIR_THRI:
                /* transmit hold register empty interrupt */
                if(po == pi)
                    break; /* buffer empty */

                /* keep stuffing character, while there's room */
                while(po != pi &&
                    inb(PORT(UART_LSR))&UART_LSR_THRE){
                    outb(buf[po],PORT(UART_TX));
                    po = (po++)%16;
                }
                msk |= UART_IER_THRI; /* add xmit interrupt */
                break;
            default:
                break;
        }
    }
    outb(msk,PORT(UART_IER)); /* go on */
}

/* set up UART */
int init_module(void)
{
    int div;
    div = 115000/9600; /* we want 9600 */
    request_RTirq(4, intr_handler); /* interrupt */
    pi = po = 0; /* buffer reset */

    outb(0x00,PORT(UART_IER)); /* clear interrupt enables */
    outb(0x00,PORT(UART_FCR)); /* NO FIFO Control */
    outb(UART_LCR_DLAB,PORT(UART_LCR)); /* set divisor address latch */
    /* set data rate */
    outb(div&0xff,PORT(UART_DLL)); /* low byte */
    outb(div>>8,PORT(UART_DLM)); /* high byte */
    outb(UART_MCR_DTR|UART_MCR_RTS,PORT(UART_MCR)); /* DTR/RTS = 1 */
    outb(MOD_8N2,PORT(UART_LCR)); /* 8 bit, 2 stop, no parity */
    outb(UART_IER_RDI,PORT(UART_IER)); /* let'er rip */

    return 0;
}
/* remove handle and turn off interrupt when done */
void cleanup_module(void)
{
    outb(0x00,PORT(UART_MCR)); /* DTR/RTS = 0 */
    outb(0x00,PORT(UART_IER)); /* stop */
    free_RTirq(4);
}

```

cost of this abstraction is that it might not always be the best interface for your application. In particular, you have no control over the latency from the time the character was received until your application gets to use it.

RT-Linux, the real-time extension to Linux, does not have a serial port driver. If you want to talk to serial ports within RT-Linux and you want exact control over your serial port, you need to write a driver for it.

Listing 3 shows a simple RT-Linux driver. The routine `init_module` installs an interrupt handler (`intr_handler`) and initializes the UART. The interrupt handler does all the work. It looks at the IIR and dispatches based on the interrupt source.

In this example, I only consider the receive and transmit interrupts. On receive, we store the character into a small circular buffer, and the transmit interrupt drains this buffer by transmitting the characters. It's a simple echo program.

One gotcha that you'll have to deal with in a serial-based interrupt handler is the following: when you want to transmit for the first time, the transmit interrupt will be off. So, you need some way to prime the transmit handler.

Typically, this situation is handled by pretending we just had a transmit interrupt and calling the handler. In our case, if the receiver gets a character, it drops through to the transmit handler to see if it needs to transmit anything, which it will the first time it's called.

This generic example would be easy to port to different environments. It only needs a routine from the environment to install an interrupt handler. Most RTOSs have that.

SCOPING OUT

What can we hook up to this? I mentioned the SX Blitz programmer from Parallax, or you could hook up a GPS receiver.

One interesting implementation is BitScope. BitScope is a digital oscilloscope module based on a PIC chip. In fact, it was one of the Design98 contest winners. BitScope uses a protocol similar to the one for the SX Blitz. You send commands to it and retrieve data via serial ports.

Building your own peripheral isn't as hard as you may think. One of the easiest ways to interface something to your PC is to use a Basic Stamp or a PicStic.

These devices (both based on PICs) are programmed in BASIC and have

serial routines for transmitting and receiving serial data. You simply hook up the hardware you want to control and write a small BASIC program.

For higher performance and lower cost, you can also use a PIC/Scenix or other 8-bit micro. There are libraries and plenty of examples for doing serial communications with these. A rate of 115 kbps is nothing for a little \$3 PIC processor.

You can also build hardware-based UARTs with discrete components (about ten chips) or with FPGAs and CPLDs. In fact, you can get predesigned cores for FPGAs from a variety of vendors.

Many times, building a serial-based peripheral is much less trouble than building a custom ISA- or PCI-bus card. It's also easier to write host software for serial ports and then write a device driver for a custom board. That's especially true for OSs like Windows NT. [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.

SOFTWARE

Software for this article may be downloaded from the [Circuit Cellar](#) web site.

REFERENCES

Linux serial how-to, <http://sunsite.unc.edu/LDP/HOWTO/Serial-Programming-HOWTO.html>
Serial-port resources, www.lvr.com/serport.htm

SOURCES

UART cores

Xilinx
(510) 600-8750
Fax: (408) 559-7114
www.xilinx.com/products/logiccore/tbl_base_lv1.htm#uart

Altera
(408) 544-7144
Fax: (408) 544-6403
www.altera.com/html/tools/megacore.html

BitScope

BitScope Designs
info@bitscope.com
www.bitscope.com

SX Blitz

Basic Stamp
Parallax, Inc.
(916) 624-8333
Fax: (916) 624-8003
www.parallaxinc.com

PicStic

Micromint
(407) 262-0066
Fax: (407) 262-0069
www.micromint.com

Sending a DOS Stamp Airmail

Ever wonder why the air inside airplanes is so dry? Neither did Fred, until he was asked to design a system to increase the humidity in the cockpit of a 737. Fortunately, a DOS Stamp gets this embedded system off the ground.

Being a Florida boy, humidity is not one of the things I'm short on. But did you know that many of today's (and yesterday's) sophisticated commercial jet aircraft don't humidify the cockpit or passenger cabin air during flight? I didn't know that either until I was chosen to put the brains behind a humidity system for a privately owned Boeing 737.

The reason for this lack of high-altitude "wet" air stems from research on aircraft-cabin humidity indicating that high cabin humidity levels shorten the life of the aircraft's interior components. This particular study also found that uncontrolled humidity systems within the cabin cause water to collect in parts of the plane that could lead to airframe and avionics failures over time.

In fact, this report is flat against installing any kind of in-flight humidity system. But, further on, the report states that if you feel you must do this, the recommendation is that the

aircraft's humidity system should be controlled and monitored by an intelligent electronic device.

Hmm...an "intelligent electronic device." Sounds like "embedded PC" to me.

Of course, this job has to be finished and ready to install in two weeks. Again,

the law of embedded-PC physics as applied to job deadlines is upheld. For the uninitiated, this law states that "The time allotted for an embedded design is inversely proportional to the complexity of the task."

Obviously, there's no time to lollygag around. I know I'll be controlling valves and motors. I also know I'll be interfacing with humidity and temperature sensors.

Although the FAA still has to bless the hardware, this system isn't a critical flight component and the job of the embedded PC isn't numerically intensive or time constrained. Therefore, I won't need a high-end multitasking embedded OS or a fancy high-priced piece of embedded hardware.

Honestly, this is a great job for a PIC. The only problem is that I'm probably going to have to fly with this thing and do some real-time in-flight tweaking. I have no problem

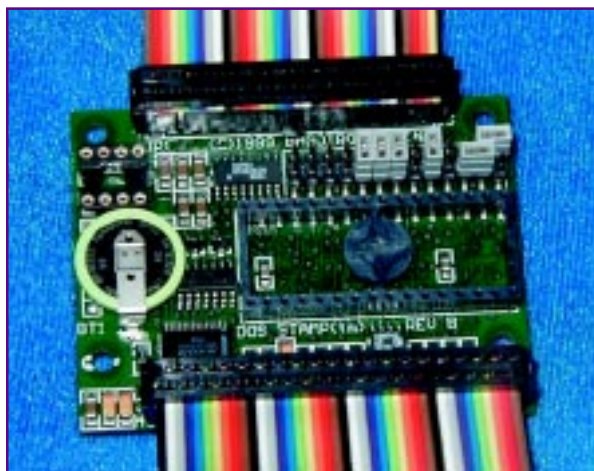


Photo 1—The DOS Stamp.... Just when you thought you couldn't afford to do an embedded DOS project.

a) Memory Map		
00000–7FFFF	SRAM	
80000–BFFFF	Flash disk socket	
C0000–FFFFF	Bootflash	
b) I/O Map		
0800–08FF	*PCS0	
0900–09FF	*PCS1	
0B00–0BFF	*PCS3	
0D00–0DFF	Real-time clock	
0E00–0EFF	ADC	
FF00–FFFF	Am188ES peripheral control block	

Table 1—Here’s a look at the SBI signals. It can’t get much simpler than this.

with flying my laptop and a PIC programmer, but it’s certainly easier to fly my laptop and download minor code changes to some sort of embedded target that only needs a quick reboot to effect the changes.

For this project, I needed a tricky little embedded widget that’s cheap and low on the learning curve. At the time, I still had PIC on the brain and I was considering using one of the many Stamp products I saw advertised in *Circuit Cellar*. Scanning the ads for the keyword “Stamp,” I found the most peculiar Stamp I’d ever seen.

It runs DOS! A DOS Stamp! Whoa, Nelly! “This has potential,” I said to myself. Moments later, I was on and off the phone with Ivan at Bagotronix, arranging to have a DOS Stamp airlifted to me.

STAMP OF APPROVAL

As its name implies, DOS Stamp is a 2.6”x2.0”x0.625” Am188ES-based embedded briquette. In addition to a built-in General Software BIOS and DOS operating system, the DOS Stamp has its own expandable I/O subsystem. Mine came with the optional 12-bit, 8-channel Maxim Max197 ADC. My application needs to access sensors, so the ADC is a nice feature. On the digital side, there are 16 general-purpose I/O lines.

You can get as tricky as DOS will let you when it comes to programming the DOS Stamp. Borland’s C/C++, Bill’s C/Visual C++, Bill’s Quick-BASIC (compiled only, please), Bob Zale’s PowerBASIC, and just about anybody else’s DOS assemblers and compilers will

make acceptable .EXE bit patterns for the DOS Stamp.

You can stuff those executable bit patterns into its 512 KB of SRAM and 128 KB of flash memory. If that’s not enough storage, it also has a socket for mounting up to 144 MB of DiskOnChip2000.

If you plan to use Bill’s or Bob’s BASIC, there’s a gotcha you should note. It seems that the Am188ES peripherals want to see *word-length* I/O reads and writes. Both Bob and Bill’s BASICs employ *byte-length* I/O operations.

Not to worry. A tight little TSR program called QBHELPER.EXE is included with the DOS Stamp software suite.

QBHELPER.EXE is called through INT 0xF4 to perform the word-long Am188ES I/O reads and writes. You only need to run QBHELPER.EXE before executing a compiled BASIC application that needs to do digital I/O operations.

There’s no Ethernet port (yet?), but the DOS Stamp sports a couple of async COM ports that can handle RS-232 and RS-485. DOS Stamp has a real-time clock with timed powerup for time-critical applications that need it.

One additional timer/counter is available to the programmer for use as desired. A watchdog timer and a power monitor provide crash protection. Note: the word “crash” here refers solely to the DOS Stamp software, not the aircraft.

The DOS Stamp comes with disks containing embedded DOS-ROM, utilities, library code, and example code. Paper documentation with color pictures is also part of the standard package.

If you’re not familiar with the Am188ES, download the user’s manual and datasheet from the AMD web site. Although Ivan did most of the up-front processor register definition work in the example code, having the Am188ES reference data helps you understand where all this stuff came from.

The DOS Stamp starter kit comes with everything an engineer needs to develop an embedded application. All I need to provide is a PC with a working serial port and my DOS compiler of choice.

This is my first experience with a DOS Stamp. So, before we jump into how to add moisture to airplane air, let’s lick the stamp and get it on its way.

LICKING THE DOS STAMP

I bet some of you will find a use for the DOS Stamp in your projects, so I’m going to take you through the process of setting it up out of the box. That way, you don’t have to repeat my errors.

Photo 1 shows my DOS Stamp. Initial hookup was a snap. I simply connected the included SBI (simple bus interface)/ADC/power ribbon-cable assembly to

the DOS Stamp, attached the serial crossover cable between the PC and DOS Stamp nine-pin connectors, and applied +5 VDC.

The pictures in the documentation were excellent, so it was easy to see what was what. All of the hardware and cables for the DOS Stamp came with the starter kit.

Now that I know how to get screenshots from NT, I decided to host this application development process on a Windows NT workstation platform. I used Bob Zale’s super-efficient space-stingy PowerBASIC to produce the jet plane’s .EXE images.

The DOS Stamp code examples include a terminal emulator called XLTERM, which was designed specifi-

Pin	JP2	Description
1	AD0	Address/Data bit 0 (tristate)
2	ALE	Address latch enable (output)
3	AD1	Address/Data bit 1 (tristate)
4	*RD	Read (active low)
5	AD2	Address/Data bit 2 (tristate)
6	*WR	Write (active low)
7	AD3	Address/Data bit 3 (tristate)
8	A0	Address bit 0 (output)
9	AD4	Address/Data bit 4 (tristate)
10	A1	Address bit 1 (output)
11	AD5	Address/Data bit 5 (tristate)
12	A2	Address bit 2 (output)
13	AD6	Address/Data bit 6 (tristate)
14	INT0	Interrupt 0 (input, rising edge, or high-level trigger)
15	AD7	Address/Data bit 7 (tristate)
16	INT1	Interrupt 1 (input, rising edge, or high-level trigger)
17	*RESET	Reset (output, active low)
20	*PCS0/GPIO9	Peripheral chip select 0 (output, active low, multiplexed with general-purpose I/O 9)
21	*PCS1/GPIO10	Peripheral chip select 1 (output, active low, multiplexed with general-purpose I/O 10)
29	GND	Ground (0 V)
30	VCC	System power (+5 V)

Table 2—A 40-pin interface/header assembly is supplied with the DOS Stamp starter kit to bring these signals out to the real world.

Listing 1—This listing is in QuickBASIC format. Note that the % follows the constant in QuickBASIC and precedes it in PowerBASIC.

```
' Constants for bit patterns and I/O addresses
CONST PIO12% = &H1000
CONST PIOMODE0% = &HFF70
CONST PDIRO% = &HFF72
CONST PDATA0% = &HFF74
CONST PDATA1% = &HFF7A
' Interrupt # for QBHELPER.EXE
CONST QBHELPER% = &HF4
' ADC operations
CONST ADCBASE% = &HE00
CONST ADCOT05% = 0
```

cally for the DOS Stamp, though other async terminal emulators like HyperTerminal are OK to use, too. I initially fired up my DOS Stamp with XLTERM, and it quickly took me through the init screens beyond the point I wanted to show you.

So, Photo 2 is the HyperTerminal view of the DOS Stamp power-up banner screen. I'll use the XLTERM terminal emulator for development as it includes a convenient upload/download capability. Instead of having to start a program called TRANSFER.EXE manually at both ends, XLTERM

kicks off the transfer at the DOS Stamp and PC end automatically.

FLY EMBEDDED JETS

At this point, the software is installed and the DOS Stamp is operational. The next step is to survey the Am188ES memory map and get a grip on where to put things.

The Am188ES is instruction-set compatible with the 80186 and has a memory address range of 1 MB. Table 1a shows the hexadecimal lay of the land. The boot flash contains the General Software BIOS

Listing 2—QuickBASIC needs an explicit structure to handle register operations. PowerBASIC handles register operations natively. For instance, REG %BX is PowerBASIC for REG.BX.

```
' Constants for bit patterns and I/O addresses
%PIO13 = &H2000
%PIOMODE0 = &HFF70
%PDIRO = &HFF72
%PDATA0 = &HFF74

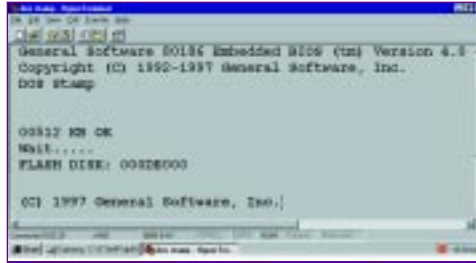
DECLARE SUB outw (outaddr AS INTEGER, outval AS INTEGER)
DECLARE FUNCTION inpw% (inaddr AS INTEGER)

'set the PIOMODE bit
x% = inpw(%PIOMODE0)
x% = x% OR %PIO13
CALL outw(%PIOMODE0, x%)
x% = inpw(%PDIRO)
x% = x% AND (NOT %PIO13)
CALL outw(%PDIRO, x%)
' clear the PIO13 data bit
x% = inpw(%PDATA0)
x% = x% AND (NOT %PIO13)
CALL outw(%PDATA0, x%)

FUNCTION inpw% (inaddr AS INTEGER)
    REG %BX , 0
    REG %DX , inaddr
    CALL INTERRUPT %QBHELPER
    inpw% = REG(%AX)
END FUNCTION

SUB outw (outaddr AS INTEGER, outval AS INTEGER)
    REG %BX , 1
    REG %DX , outaddr
    REG %AX , outval
    CALL INTERRUPT %QBHELPER
END SUB
```

Photo 2—This BIOS start-up screen is much like the ones found on desktops except for the “DOS Stamp” banner entry!



and DOS and about 106 KB of usable user flash memory. Address range 80000–BFFFF is the DiskOnChip area. The Am188ES I/O range is 64 KB wide. Table 1b shows you how it looks on the DOS Stamp.

PCSX signals are active-low Am188ES chip selects. PCS0 and PCS1 are pinned out to the DOS Stamp’s SBI connector. As you can see in Table 2, the SBI consists of data and address lines along with some read/write, address decoding, and interrupt signals. Using the SBI signals to expand I/O is clarified in Figure 1.

The DOS Stamp’s real-time clock (RTC) is really a Dallas Semiconductor DS1689. Table 2 is the complete SBI header pinout. Note that in addition to the SBI address and data stuff, there are other pins like *KICKSTART and *PWRUP.

The DOS Stamp RTC can power up itself and anything attached to the *PWRUP pin on an RTC alarm time match. Because the DS1689 keeps time regardless of

applied power, the DOS Stamp can wake up, shower and shave, set the alarm for the next wakeup call, and turn out the lights. A simple push-button switch is attached to the *KICKSTART line to manually power up the load and, in a battery-powered situation, the DOS Stamp too.

There’s nothing special about the MAX197. It’s just a great part with lots of ADC bells and whistles included as standard equipment. Ivan thoughtfully included the MAX197 datasheet.

The Am188ES’s Peripheral Control Block register map is shown in Table 4. Here’s why you should download the Am188ES user’s manual and datasheet.

Listing 1 is a snippet of some of the DOS Stamp example code. Note the I/O addresses in Listing 1 and their counterparts in the AMD data document represented in Table 4. You need to know this stuff to successfully deploy the DOS Stamp.

It’s time to toggle bits and read sensors. The first order of business is to establish that PowerBASIC is indeed compatible with the QBHELPER.EXE module.

Since I’m in the dark without a lantern here, I’m going to start with a known. There shouldn’t be any problems, but this is my first experience with this module.

The DOS Stamp starter kit contains a file set called DSHELL0. DSHELL0 is shipped in QuickBASIC source format and comes ready to roll as an executable. I ran the QBHELPER first and then DSHELL0.EXE to establish a baseline for my conversion.

Basically, DSHELL0 checks some environment settings, turns on COM ports, does A/D conversions, and toggles bits on the Am188ES’s I/O port. DSHELL0.EXE ran flawlessly.

Time to start the QuickBASIC to PowerBASIC conversion. As far as source files are concerned, QuickBASIC and PowerBASIC don’t always agree. So, I ran the QB2PB (QuickBASIC to PowerBASIC) module that ships with PowerBASIC.

I was hoping to have the QuickBASIC source code magically altered and served as PowerBASIC source, but the only good thing the “conversion” program did for me was point out what I needed to change.

The DSHELL0 program begins by attempting to discover if the hardware is a PC or the DOS Stamp. Following several iterations of conversion and code substitu-

Pin	JP2	Description
1	AD0	Address/Data bit 0
2	ALE	Address Latch Enable
3	AD1	Address/Data bit 1
4	*RD	Read Strobe
5	AD2	Address/Data bit 2
6	*WR	Write Strobe
7	AD3	Address/Data bit 3
8	A0	Address bit 0
9	AD4	Address/Data bit 4
10	A1	Address bit 1
11	AD5	Address/Data bit 5
12	A2	Address bit 2
13	AD6	Address/Data bit 6
14	INT0	Interrupt Request input 0
15	AD7	Address/Data bit 7
16	INT1	Interrupt request input 1
17	*RESET	Reset output, active low
18	*EXTRESET	External reset input, active low
19	*KICKSTART	external startup signal, active low
20	*PCS0	Peripheral Chip Select 0 output, active low
21	*PCS1	Peripheral Chip Select 1 output, active low
22	*PWRUP	Powerup output, active low, open drain
23	TXRX1+	COM2 RS-485 transmit/receive +, input/output
24	TXRX1-	COM2 RS-485 transmit/receive -, input/output
25	RX0	COM1 RS-232 receive input
26	RTS0/TX1	COM1 RS-232 request to send output / COM2 RS-232 transmit output, function depends on JP3 jumper selection
27	TX0	COM1 RS-232 transmit output
28	CTS0/RX1	COM1 RS-232 clear to send input / COM2 RS-232 receive input, function depends on JP3 jumper selection
29	Ground	
30	+5 V	
31	Ground	
32	VREF	ADC voltage reference output, 4.095 V
33	AI0	ADC Input 0
34	AI1	ADC Input 1
35	AI2	ADC Input 2
36	AI3	ADC Input 3
37	AI4	ADC Input 4
38	AI5	ADC Input 5
39	AI6	ADC Input 6
40	AI7	ADC Input 7

Table 3—This register set is the parallel of the TRIS register set in the PIC world.

tion, I never got PowerBASIC to execute this module correctly.

After a few hours, I decided that I knew which machine my program was executing on and that was good enough. I moved on to higher priority tasks. I decided to attack the bit I/O area first.

Listing 2 is PowerBASIC source code that implements bit-oriented I/O operations. The heart of the bit I/O transfer is contained within a couple of functions, `inpw` and `outw`.

Judging by what the `inpw` and `outw` functions are asking for, I think it's safe to assume that some very low-level activity involving loading registers and calling BIOS services is taking place here.

This whole I/O process makes a bit more sense if you identify the players up front. The Am 188ES datasheet spells out how the `PIOMODE0` and `PDIR0` values play into the I/O mix.

The constant `PIOMODE0` references the physical Am188ES PIO Mode 0 register. Each bit position in the PIO Mode 0 register corresponds to one of the Am188ES's 16 GPIO (general-purpose I/O) lines. The PIO Direction registers correspond bit by bit with the PIO Mode registers.

There are four settings for each GPIO bit depending on how you set the PIO Mode and PIO Direction bits. My code in Listing 2 is interested in PIO bit 13, which equates to GPIO 2.

Another register set, PIO Data, shadows the PIO Mode and PIO Direction bits. If an I/O pin is enabled as an output, the

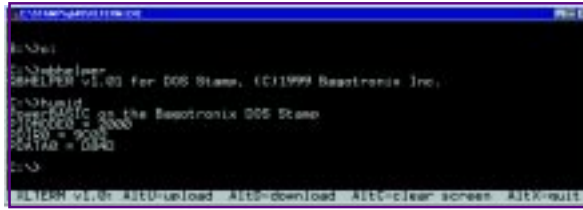


Photo 3—By the way, the LED tied across +5 V and GPIO 2 did illuminate.

value in the corresponding bit of the PIO Data register is driven to the GPIO pin. For pins selected as inputs, the value of the pin is represented by the value in the corresponding PIO Data register bit.

Now that all of the registers and values involved have been defined and identi-

fied, the next step is to twiddle the bits. To configure PIO13 (GPIO 2) as an output, we simply set the PIO Mode bit and clear the PIO Direction bit for GPIO 2. You can't see the state of GPIO 2 or the LED I tied to the GPIO 2 pin because you're not here. So, I'll take GPIO 2 low and print the hex

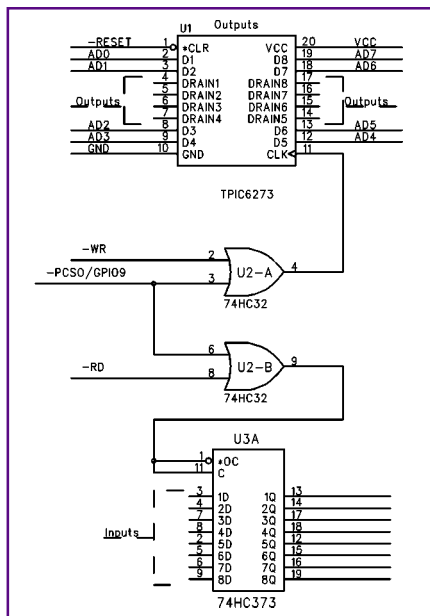


Figure 1—A picture is worth 0x3E8 words.

value so you can see the correlation of the bits between the PIO Mode, Direction, and Data registers. Photo 3 is the output data captured from the DOS Stamp's registers.

That takes care of any valves, pumps, compressors, or switches I'll have to diddle with. Let's attach a sensor to the DOS Stamp.

The sensor of choice here is HyCal Engineering's Survivor II relative-humidity active sensor. The Survivor II is a CMOS integrated circuit with a thin-film RH sensor imbedded into its monolithic structure.

The beauty of this device is that it only needs a standard +5-VDC supply and it outputs a voltage that is proportional to the surrounding relative humidity. It is fully calibrated and comes with factory-provided reference-voltage readings for 0% and 75.3% relative humidity.

The Survivor is linear so it's easy to convert the voltages to humidity. My sensor reads 0.669 V at 0% and 2.922 V at 75.3%.

The Survivor II is aptly named, as it is enclosed in a TO-5 can with a hydrophobic filter. I misplaced the sensor on my bench and found it unharmed, jammed under a keyboard. There's also a 100-k Ω platinum thermistor inside the TO-5 case that is pinned out for temperature measurement.

I kind of pooh-poohed the MAX197 as just one of the boys earlier. Sorry. Actually the MAX197 is a dream to work with. As with the digital I/O code, again, player identification is the key.

The DOS Stamp's base address for the MAX197 is defined as 0xE00. A conversion is initiated by simply writing a byte containing the desired analog input channel, the range and polarity, and clock mode to the base address.

End of conversion is sensed as a logical low at GPIO 11 (PIO18). The digital representation of the sensor analog output is then read into the DOS Stamp using the MAX197's base address data lines.

That's it. the acquired voltage (humidity in our case) is calculated and displayed. It's all laid out in Listing 3.

AIRMAIL STAMP

The high-flying humidity system that this Stamp will stick to is basically a closed-loop system consisting of a water tank, a mist engine (compressor and mist nozzles), temperature and humidity sensors,

solid-state relays, valves, and a cockpit control panel.

To effect the 737 humidifier system, the DOS Stamp must perform two basic functions. It must sense both the cabin and passenger compartment humidity envi-

ronments and be able to control them. I've shown you how those two humidity control goals can be easily achieved using simple programming, some simple logic, simple programming algorithms, off-the-shelf hardware, and a DOS Stamp.

Listing 3—The Circuit Cellar Florida Room came in at 1630 steps or 1.99 V of humidity, which equates to about 44%.

```

DECLARE SUB outw (outaddr AS INTEGER, outval AS INTEGER)
DECLARE FUNCTION inpw% (inaddr AS INTEGER)
DECLARE SUB ADCput (channel AS INTEGER, range AS INTEGER)
DECLARE FUNCTION ADCready% ()
DECLARE FUNCTION ADCget% ()

#include "C:\PB35\PB35.INC"

%ADCBASE = &HE00
%ADCOT05 = 0

I% = 0
DO WHILE INKEY$ = ""
    CALL ADCput(I%, %ADCOT05)
    DO WHILE NOT ADCready%
        LOOP
        j% = ADCget%
        ' .669 V is relative 0% for sensor
        ' .0012207 is 1 digital step
        ' .029 V is 1% of humidity on a 5-V scale
        PRINT #1, "HUMIDITY IS ";((j% * .0012207) - .669) / .029; "%"
        ' delay
        FOR f = 0 TO 10000 STEP .5
            NEXT f
        LOOP
    system

FUNCTION ADCget%
    I% = inpw%(%ADCBASE)
    ADCget% = I%
END FUNCTION

SUB ADCput (channel AS INTEGER, range AS INTEGER)
    j% = channel OR range OR &H40
    OUT %ADCBASE, j%
END SUB

FUNCTION ADCready%
    ' returns 0 if ADC not ready, non-zero if ADC is ready
    ' check status of PIO18 pin
    ' if low, data is ready
    I% = inpw%(%PDATA1) AND &H4
    IF I% <> 0 THEN
        I% = 0
    ELSE
        I% = -1
    END IF
    ADCready% = I%
END FUNCTION

FUNCTION inpw% (inaddr AS INTEGER)
    REG %BX , 0
    REG %DX , inaddr
    CALL INTERRUPT %QBHELPER
    inpw% = REG(%AX)
END FUNCTION

SUB outw (outaddr AS INTEGER, outval AS INTEGER)
    REG %BX , 1
    REG %DX , outaddr
    REG %AX , outval
    CALL INTERRUPT %QBHELPER
END SUB

```

PIO data 1 register	7Ah	11-5
PIO direction 1 register	78h	11-4
PIO mode 1 register	76h	11-3
PIO data 0 register	74h	11-5
PIO direction 0 register	72h	11-4
PIO mode 0 register	70h	11-3

Table 4—The Am188ES's Peripheral Control Block register map. Get the Am188ES datasheet for the full meaning.

I'm out of paper and my mouth is kinda dry. If you see a DOS Stamp in your future and need to know more about it, all of the DOS Stamp manuals and reference material I mentioned can be had as Acrobat PDF files on the Bagotronix web site. There are plenty of detailed photos there, too. Ivan is available via e-mail or phone for any technical question you may have.

So, the next time you're offered that complimentary beverage at 30-odd thousand feet, you'll know that on top of being nice, the airline doesn't want you to dry up! Thanks to Ivan at Bagotronix as he and the DOS Stamp have again proven that it doesn't have to be complicated (or dry) 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

DOS Stamp

Bagotronix, Inc.
(850) 942-7905
Fax: (850) 942-7905
www.bagotronix.com

Am188ES

Advanced Micro Devices, Inc.
(408) 732-2400
Fax: (408) 732-7216
www.amd.com

MAX197

Maxim Integrated Products
(408) 737-7600
Fax: (408) 737-7194
www.maxim-ic.com

PowerBASIC

PowerBASIC, Inc.
(800) 780-7707
(831) 659-8000
Fax: (831) 659-8008
www.powerbasic.com

QuickBASIC

Microsoft Corp.
(206) 882-8080
Fax: (206) 936-7329
www.microsoft.com

Survivor II

HyCal Engineering
(818) 444-4000
Fax: (818) 444-1314
www.hycalnet.com

FEATURE ARTICLE

Hari Ramachandran

IrDA Technology

Part 2: Protocol Layers

In the second half of this series, Hari explains, what he feels, is the most daunting step of working with Infrared Data Association (IrDA) solutions—implementing the protocols. Pack your bags, you're on your way to understanding IrDA.



In Part 1, I presented an overview of putting together an IrDA solution, focusing mainly on the IrDA hardware. Once the basic hardware is running, your next step is probably the most daunting—getting the IrDA protocols implemented.

My experience over the last couple of years has been that developers shy away from integrating IrDA solutions into products because of the perceived complexity of the IrDA protocol. But once you understand the concepts, you'll be in a better position to evaluate the options open to you—either developing the stack from scratch or purchasing a commercial IrDA protocol stack.

In this article, I want to guide you along the whole process, explain key concepts, and introduce you to the IrDA documentation as well as a utility that will get you started with a hands-on evaluation of the protocol.

WHAT YOU'LL NEED

The first thing you need to do is download these documents from the IrDA web site:

- Serial Infrared Link Access Protocol (IrLAP) V.1.1, June 16, 1996
- Infrared Link Management Protocol (IrLMP) V.1.1, January 23, 1996

To get hands-on experience, also download the Parallax Research IrLAMP software from the *Circuit Cellar* web site. This software is a minimal implementation of the IrLAP and IrLMP software with some diagnostics and logging capabilities that will enable you to experiment and understand key aspects of the IrLAP and IrLMP stacks.

GETTING THE MACRO VIEW

You may want to review the protocol stacks and layers (see Figure 1, *Circuit Cellar* 111, p. 60). I discussed the physical layer last month.

To cover all aspects of the protocol and application layers would be a bit much for a magazine article. So, here I want to focus instead on the IrLAP, IrLMP, and IrLMP-IAS protocols.

These protocols constitute the minimal functionality required to build a working IrDA link. To do anything useful, you need to add the other protocols on top of this set, but I'll leave that as an exercise for the reader.

We can either dive straight in, analyzing every line of the protocol documentation (not!); or we can take a narrative approach. To keep you from nodding off, I'll discuss the relevant points briefly here. I'll also tell you which document to look at for further information.

I'll start at the lower level, discuss the software required at the framer level, and then move up to the IrLAP and IrLMP protocols. After the overview, you can get your hands into the IrLAMP software.

FRAMERS AND WRAPPERS

The physical layer is a combination of hardware and firmware/software

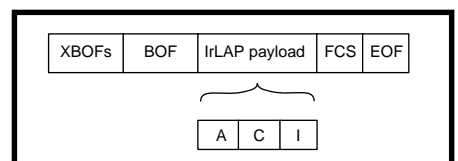


Figure 1—The IrLap Frame Structure calls for the payload to be preceded by at least one BOF

Command	Description
SNRM	Set Normal Response Mode—Once a device is identified, the primary sends this command to establish a point-to-point connection with the device. At this stage, parameters such as speed of communication are negotiated between two devices
DISC	Disconnect—terminates a connection
XID	Discovery—sent by the primary to discover all devices within range and is sent back by the secondary device as a response to XID
UA	Affirmative response to SNRM or DISC

Table 1—These basic IrLAP commands handle transmissions between IrDA devices.

that receives and sends and checks the integrity of a received IrLAP frame.

“Framer” is the unofficial term for describing the firmware or software that parses the received frame and builds up a frame to be sent out.

“Wrapper” describes the envelope into which the IrLAP frame is dropped. There are different wrappers and framers for each speed range of operation:

- SIR: 9600–115,200 bps
- MIR: 576 kbps and 1.152 Mbps
- FIR: 4 Mbps

If your hardware only supports SIR, you only need to support an SIR framer. I’ll focus on the SIR framer for now.

(Bear in mind, that even though there are different algorithms and framers for extracting the IrLAP frame for each speed range, the IrLAP payload is the same for all three framers). Detailed information about the framers is in Appendix D of the IrLAP documentation.

An SIR frame is shown in Figure 1. XBOFs (extended beginning of frames) is a series of 0xFF or 0xC0 characters preceding the IrLAP payload. The IrLAP payload must be preceded with at least a single BOF (0xC0).

The number of XBOFs is dictated by the IrDA protocol. Generally, for slower devices, more XBOFs are generated to prepare the UART or micro-controller for the actual payload.

The IrLAP payload consists of the address (A), control (C), and information (I) fields. These fields are described in the IrLAP documentation.

To ensure the integrity of the sent data, a two-byte FCS (Frame Check Sequence) field is generated and transmitted along with the data. IrLAP uses the CRC-CCITT check method. The algorithm/code for generating this checksum is in Appendix D of the IrLAP documentation.

When a frame is sent, the FCS is computed. And when the frame is received, the FCS is calculated and compared against the transmitted

FCS. If there’s a discrepancy, the frame is corrupt and should be abandoned. Finally, the EOF character determines the end of the frame (0xC1).

Generally, the SIR framer operates on a byte-by-byte basis, receiving each byte and determining if a complete frame was received. Because the algorithm for detecting the whole frame relies on BOFs and EOFs to ascertain the payload and FCS, these characters can’t appear in the payload or the FCS.

A simple set of “transparency” rules takes care of this condition, during transmission as well as reception. Appendix D of the IrLAP documentation has the details.

IR LINK ACCESS PROTOCOL

Once a valid IrLAP frame is received by the framer, it’s handed to IrLAP. IrLAP is a robust, frame-based protocol for connecting and exchanging data between IrDA devices.

IrLAP began as a variant of IBM’s HDLC protocol, with modifications made to take into account the ad hoc nature of IR connections. Unlike wired connections, the configuration of an IrDA link is dynamic.

Devices can be brought within range and just as easily moved out of range. Multiple devices can be within range and must be correctly identified and managed.

IrLAP is responsible for:

- discovering all devices within range
- reporting possible connection devices to higher layers
- establishing connections between one devices
- connecting and maintaining a communication session, which includes ensuring data integrity and proper process for clean disconnects of interrupted sessions and timeouts
- support for disconnecting from devices once data transfer has terminated

So how does IrLAP work? The first step in the IrLAP connection process is for one

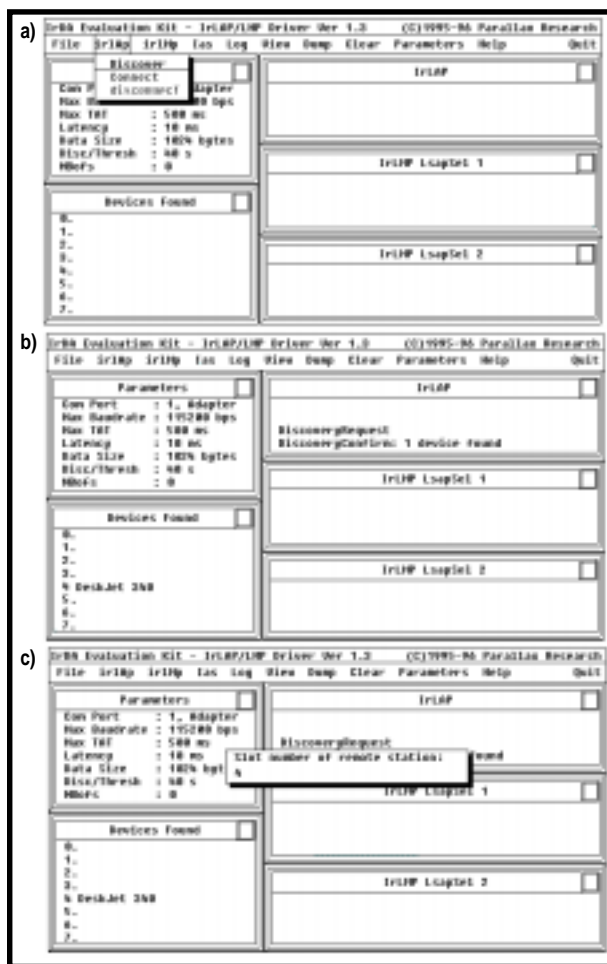


Photo 1a—The IrLAMP software is a simple way to start experimenting with the IrDA protocol. b—The software has discovered an HP Deskjet 340 IrDA printer within range. c—IrLAP protocol is a point to point protocol, so you can discover multiple devices, but you can only connect to one at a time.

device—the primary—to initiate a discovery operation.

The discovery process involves sending commands to all IrDA devices within range. This operation is usually initiated by the user or by the device when it's ready to transfer data.

To initiate the discovery operation, the primary sends out the XID command frame. Any devices in range respond with an indication and identify themselves to the primary.

Multiple devices can be discovered so IrLAMP forces each device to respond with a unique timeslot number. The IrLAMP specs outline methodologies and algorithms to resolve contention situations.

Once a discovery operation is completed, the primary has a list of devices it can connect to either under user intervention or through some logic within the primary. The primary initiates a connection to a device by sending the SNRM (Set Normal Response Mode) command.

Once a successful connection is established, the device that was connected is called the secondary. Even though there may be multiple devices out there, SNRM forces a connect to a specific device.

Only that connection is active (i.e., a unique data link is established between the primary and the secondary). Other discovered devices remain dormant.

Once connected, the primary and secondary exchange information. When the primary finishes, it initiates a disconnect message.

Because the IR link is ad hoc in nature (a device can be moved away during the transmission), various timeout procedures and checks for connection integrity (time based) are constantly exercised during the connection.

Some relevant IrLAMP commands are given in Table 1. Please refer to the IrLAMP documentation for further details.

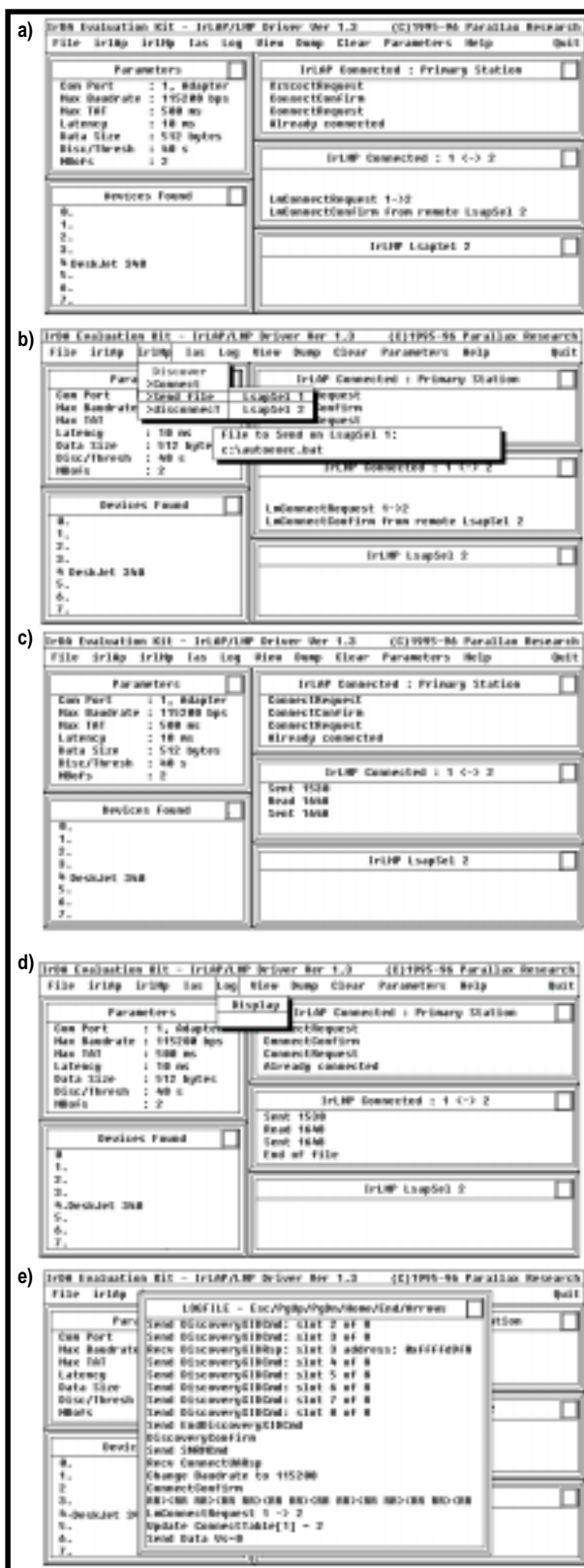


Photo 2a—You can see a successful LM connection between LSAP1 (IrLAMP's channel) and LSAP2 (on the Deskjet 340). Most printers support printing services on LSAP 2. **b**—Once a channel connection is established, you can send a file across the channel. In this case, select LSAP1 (which is connected to LSAP2 on the printer), specify a file and away you go! **c**—The IrLAMP1 window displays the progress of the data transmission. **d**—IrLAMP supports a simple log file that enables you to trace the progress of the IrDA session. To activate this, select Log->Display. **e**—You can trace what happens from the Discovery procedure all the way to the connect, data transfer, and disconnect.

IR LINK MANAGEMENT PROTOCOL

Think of IrLAMP as a glorified wire. Once an IrLAMP connection is established, data needs to be routed to specific endpoints. What's an endpoint?

If you built a multifunction IrDA device that supports IR connectivity to a printer, fax, or LAN connection, then each application can be thought of as a distinct endpoint.

The IrLAMP layer receives data for endpoints but can't differentiate between applications. It hands the responsibility of routing the data to IrLMP.

IrLMP allows for multiple channels of information through a single IrLAMP channel so more than one channel of information can be open at any time. This way, multiple apps within two communicating IrDA devices can be active, each with its own channel.

IrLMP permits routing of information from one application to another. It's kind of like a mailman, routing messages to the correct addressee.

Figure 2 shows how routing is handled? The address and control fields are IrLAMP-specific fields. The I-frame, however, contains actual data or commands that encapsulate IrLMP commands and/or data.

The IrLMP command/data frame is called an LmPDU (link management protocol data unit). The channels of connectivity are called logical selection access points (Lsaps).

The first two fields of the LmPDU contain the destination and source Lsap address. The DlsapSel field (destination Lsap) is the channel number of the endpoint that is to receive the PDU. The SlsapSel field (source Lsap) is the channel number of the endpoint that's sending the PDU.

Routines in the IrLMP module receive IrLAMP frames and route the PDU to the appropriate

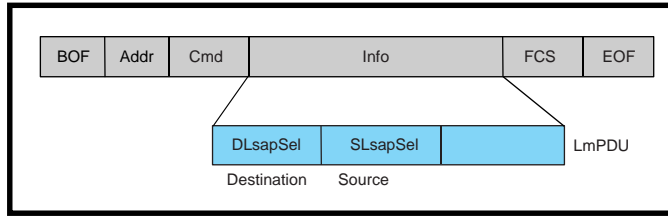


Figure 2—LmPDUs are embedded in IrLAP frames and contain actual data or commands.

ate endpoint. The LmPDU can consist of commands or data.

Before all these intricate links can be formed, there must be a way a device can query another device to find out what services it provides and which Lsaps provide those services.

For example, if a Windows CE machine wants to connect to an IrDA printer, it needs to query the printer to determine which Lsap it should connect to (i.e., which Lsap provides the print services?).

This mechanism is provided by the IAS (information access service), which is part of the IrLMP specification. As I mentioned in Part 1, the IAS is like a Yellow Pages of services provided by an IrDA device.

Lsap 0 is the IAS Lsap. So, a device wanting to initiate an IAS query needs to connect to Lsap 0, query the device, and disconnect from Lsap 0 before establishing another connection.

The IAS is controlled and managed through the IAP (information access protocol). Figure 3 outlines the structure of the IAP frame, which is similar to the IrLMP frame but subdivided.

GET YOUR FEET WET

If you're anything like me, all this talk has probably put you in a state of near catatonia. I bet you're itching to see some of these concepts in action.

The IrLAMP software is a good way of getting started and understanding key IrDA concepts. It mostly focuses on the IrLAP, IrLMP, and IAS concepts.

Using IrLAMP, you can initiate an IrLAP discovery, identify devices within range, do an IAS query, and transfer data to a specific LSAP.

The setup instructions are in the PDF file. Ideally, you should use two PCs running IrLAMP, but you can use an IrDA printer (e.g., an HP Laserjet 5P or Deskjet 340) with the software to initiate an IrDA print session.

USING IrLAMP

On startup, you should see the screen in Photo 1a. The key features of the user interface are the Channel windows, comprising the IrLAP channel and two IrLMP channels.

Note that the term "IrLMP channel" is the same as LsapSel (link service access point selector) term used in the IrLMP document. Any activities related to these channels are reflected in this window.

The Parameters window reflects the quality of service (QOS) settings for the link. These parameters can be modified by the user.

The Devices Found window is updated during a Discovery operation. Any responding devices that can connect are displayed here. The next step is to discover devices within range.

Before proceeding, ensure that another PC running IrLAMP (or as in this example, a Deskjet 340 or HP Laserjet 5P IrDA printer) is in range.

Selecting IrLAP→Discover causes DiscoverXID to be sent out, which requests any devices in range to identify themselves. IrLAMP displays the device ID in the timeslot that the IrDA device chose to respond in.

Photo 1b shows a positive response to the IrLAP→Discover sequence. The IrLAP Channel window describes the activity in the IrLAP channel. It indicates that a device has responded to

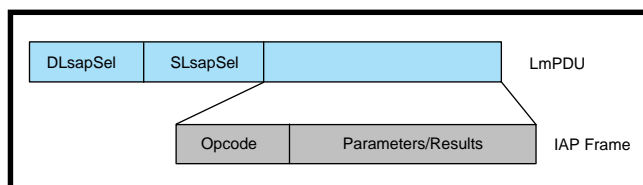


Figure 3—Link management frames are routed to specific logical channels.

DiscoverXID (the Discovery Confirm message is displayed).

In the Devices Found window, the device that responded is displayed. In this case, it's a Deskjet 340 IrDA printer.

Slot 4 was allocated for communications to this device. The next step is to establish an IrLAP connection.

Photo 1c shows this sequence of events. Select IrLap→Connect, and you'll be prompted for which device slot you'd like to connect to.

The IrLAP window displays a confirmation of the request. The next step is to establish an IrLMP connection.

The IrLAMP software supports two channels—Lsap1 and Lsap2. Select IrLmp→Connect→LsapSel 1 to 2. The IrLMP window updates with the results of this request (see Photo 2a).

You've now established an IrLMP connection between Lsap1 and Lsap2. To send some data across, follow the steps shown in Photo 2b.

Enter the file name you want to transfer. You'll see data-transfer activity in the IrLMP Channel 1 window. On successfully transferring data, a message appears in the IrLMP Channel 1 window (see Photo 2c).

If you connected to a IrDA printer, you'd see the printed output. Keep in mind that both the LaserJet 5P and Deskjet 340 accept output through Lsap2, so connect to that channel to print the file.

If you connected to another PC running IrLAMP, you can view the contents of the transferred text file by selecting View (on the second test PC). If you sent a binary file, use Dump to view the contents of the file.

You can always view the series of IrDA events by viewing the log. Just follow the sequence in Photos 2d and 2e.

The logfile describes the series of events/commands exchanged during the IrDA session. In conjunction with the IrLAP and IrLMP documentation, this log gives you a good feel for the series of events involved in establishing and maintaining an IrDA link.

WHERE TO NEXT

Well, now that you have some level of theoretical knowledge of the IrDA protocol stack, you're in a better

position to decide whether you're interested in developing the IrDA protocol stack yourself or just purchasing it. If you're looking to buy, CounterPoint Systems Foundry has a robust stack and add-on modules that is currently used in a number of commercial applications.

IrDA is supported under Linux as well. You can log onto the Linux web site to find out more about this.

And if you are interested in learning how to write IrDA applications under Windows 95, 98, or 2000, head over to Microsoft's web site and do a search on IrDA. You'll find some interesting articles on building IrDA applications. ☐

Hari Ramachandran is the managing director and founder of Parallax Research, a design company focused on providing IrDA solutions. He designed the HP HSDL 7001 and HP HSDL 7000 IrDA chips, and developed IrDA protocol stack software for numerous companies and specific applications. You may reach him at hari@parallax.com.sg.

SOFTWARE

The IrLAMP software may be downloaded via the *Circuit Cellar* web site.

SOURCES

IrLAMP

Parallax Research
+65 791-7388
Fax: +65 793-0086
www.parallax-research.com

CounterPoint Systems Foundry
(541) 758-6123
Fax: (541) 758-6120
www.countersys.com

IrDA
(925) 943-6546
Fax: (925) 943-5600
www.irda.org

Linux
(301) 490-7245
Fax: (301) 490-7162
www.linux.org

Microsoft
(206) 881-8080
Fax: (206) 936-7326
www.microsoft.com

MICRO SERIES

Mark Balch

High-Definition TV

Video Formats and Transport

Part 1 of 3

You may still be watching “Seinfeld” reruns in 2006, but according to the FCC, you won’t be watching them via analog NTSC signals. Stay tuned as Mark kicks off this series on HDTV formats and signal transportation.



In recent years, high-definition television (HDTV) has been attracting lots of attention and money—and for good reason. The Federal Communications Commission has mandated that all U.S. terrestrial television stations begin broadcasting digital television (DTV) by 2002. If all goes according to plan, the broadcast of our familiar, decades-old analog NTSC signals will cease at the end of 2006.

Already, affiliates of the four major TV networks in the nation’s ten largest population centers began broadcasting DTV on May 1, 1999, in accordance with the FCC’s phased roll-out plan. These stations will be joined by their colleagues in the top 30 markets by November 1, 1999—a requirement also mandated by the FCC.

Various other stations across the U.S. have voluntarily decided to begin DTV broadcast within this initial timeframe. All other commercial stations throughout the country are to begin their DTV broadcasts by May 1, 2002.

Between now and 2006, the U.S. television industry will undergo an immense transition, the likes of which it has never experienced. Unlike the move from black and white to color

TV, DTV will not be backward-compatible with the analog NTSC equipment that has become ubiquitous. DTV requires a completely new infrastructure at all levels—studios, transmitters, and people’s homes (see the sidebar “Managing the Transition”).

In the context of terrestrial TV broadcast, the term “DTV” does not directly equate with high definition (HD) but simply refers to the MPEG-2 compressed digital transport and formatting of television pictures and their associated data.

DTV is extremely flexible because it gives broadcasters a fixed chunk of bandwidth to organize in almost any way they wish. There are stations today conducting multichannel DTV broadcasts of both HD and standard definition (SD) material.

The format of terrestrial DTV broadcast in the U.S. is largely defined by the Advanced Television Systems Committee (ATSC), an industry standards group whose membership includes broadcasters and equipment manufacturers. Many of the ATSC’s recommendations have been adopted by the FCC and are therefore mandatory in the U.S.

In a feature article a couple years ago (*Circuit Cellar* 86), Do-While Jones presented a thorough primer on the differences between HD and SD signals and provided a description of how MPEG-2 compression works.

In this article, I provide an overview of the various HD formats available and explain how uncompressed HDTV is transported along with audio and data prior to MPEG-2 encoding.

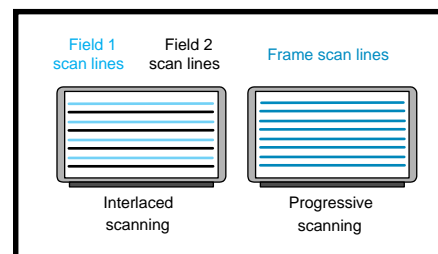


Figure 1—Interlaced scanning (used in NTSC signals) draws one field at a time on the screen, leaving room between its lines for a line from the other field. This results in the fields being temporally spaced (time shifted). In a 60-Hz field rate video format (30-Hz frame rate, assuming two fields per frame), each field represents a snapshot of the picture that is 16.67 ms after the preceding field and 16.67 ms before the next field. In contrast, progressive scanning draws the whole frame in one pass representing one snapshot in time.

The remainder of this series examines DTV on the other side of the encoder and explains how all the pieces fit together to deliver service to people's TV sets.

HD VIDEO FORMATS

The Society of Motion Picture and Television Engineers (SMPTE) has defined standard HD video scanning formats, commonly referred to as rasters. These standard rasters specify the number of pixels per horizontal

scan line, the number of scan lines per frame (or field), whether the frame is interlaced or progressively scanned, the number of frames (or fields) displayed per second, and other information required to implement a given format.

The interlaced or progressive scanning attribute refers to how the image is assembled on a TV screen. This concept is illustrated in Figure 1.

SMPTE defined two basic HD rasters: 1080 and 720 active vertical lines of video. The 1080-line format takes either an interlaced (1080I) or progressive (1080P) form. The 720-line format is defined for progressive scanning only (720P).

These HD formats use square pixels and have an overall frame aspect ratio of 16:9. Similar to movie screens, this wider image is more like the human field of vision.

In contrast, NTSC has the familiar 4:3 aspect ratio. The 1080- and 720-line formats contain 1920 and 1280 pixels, respectively, per line that are sampled at approximately 74 MHz.

Each basic raster is available in several frame rates that can be divided into two categories: normal and $1/M$ rates. Table 1 lists the different frame rates.

The normal rates are frame rates like 60, 30, 25, and 24 Hz and have associated sampling frequencies of 74.25 MHz. The $1/M$ rates are the normal rates divided by $M = 1.001$, giving them an associated sampling frequency of 74.1758 MHz.

Common Format Name	Samples per Active Line	Total Lines per Frame	Frame Rate (Hz)	Samples per Total Line
1080I	1920	1125	30	2200
1080I	1920	1125	29.97	2200
1080I	1920	1125	25	2640
1080P	1920	1125	60 ¹	2200
1080P	1920	1125	59.94 ¹	2200
1080P	1920	1125	50 ¹	2640
1080P	1920	1125	30	2200
1080P	1920	1125	29.97	2200
1080P	1920	1125	25	2640
1080P	1920	1125	24	2750
1080P	1920	1125	23.97	2750
720P	1280	750	60	1650
720P	1280	750	59.94	1650

Table 1—Each of the three high-definition scanning formats as defined by SMPTE is denoted by a different vertical and horizontal resolution as well as interlaced or progressive scanning mode. Multiple frame rates are specified for each raster. Countries with 60-Hz AC power lines generally choose the 60- or 30-Hz basic rates, while those with 50-Hz power generally choose the 50- or 25-Hz rates. ([1] Double bandwidth formats ($F_s = 148.5/148.35$ MHz)).

The $1/M$ rates exist so that legacy material can be more easily supported. NTSC has a frame rate of $30/M$ Hz, which is approximately equal to 29.97 Hz.

Table 1 also lists the corresponding samples per total line for each frame rate and raster combination. These numbers were chosen mathematically to maintain a constant sampling frequency and raw bandwidth for all HD frame-rate and raster combinations of either the normal or $1/M$ rates.

Each pixel (sample) consists of a chrominance (color) and a luminance (brightness) value. Therefore, there are twice as many data words as there are samples in a line.

Because each raster has a constant number of active pixels per line, the duration of the horizontal blanking interval (HBI) changes to make the bandwidth relationship calculate correctly. The HBI is the portion of a line that is not visible onscreen. It is the dead time in the signal that allows the electron beam within the CRT to return to the beginning of the next line.

HD SERIAL DIGITAL INTERFACE

TV signals are inherently analog, yet are often transmitted and stored digitally in studios. Digital transmission and storage provides the same signal-quality benefits to TV that it provides to all other data.

SMPTE standardized both serial and parallel digital interfaces for SD, and these have been in use for quite some time. The obvious benefit of serial over parallel is the ability to use a single coaxial cable to carry a TV signal. When SMPTE created the HD rasters, they realized the corresponding need for an HD digital interface. The 292M standard was the result.

SMPTE 292M is a 1.485-Gbps serial digital interface (SDI) that is

similar to the SD version, 259M, which most commonly runs at 270 Mbps. Most of 259M's basic structure is reused, and some new and useful features have been added. 292M is specified at 1.485 Gbps to allow the carriage of pixels sampled at 74.25 MHz with 10-bit resolution.

Recall that each pixel consists of chroma and luma sample (S):

$$\frac{7.425 \times 10^7 \text{ pixels}}{s} \times \frac{2 S}{\text{pixel} \times 10 \text{ bpS}} = 1.485 \times 10^9 \text{ bps}$$

But what about the $1/M$ rates, you ask? I'm warning you: Don't ask a question if you don't want to hear the answer! SMPTE decided to change the clock rate for the interface as well.

Therefore, for the $1/M$ rates, everything about 292M SDI is the same except the data rate is now divided by M and equals roughly 1.4835 Gbps. Therefore, either the clocks in an HD

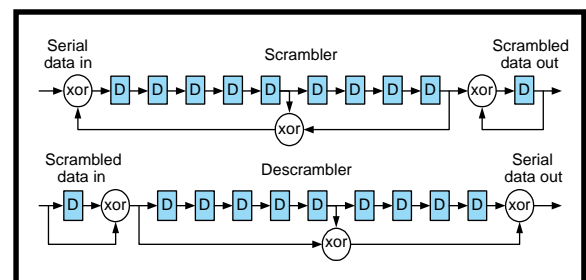


Figure 2—SMPTE provides sample implementations for serial scrambling/descrambling that meet the 259M and 292M requirements. These schematics contain two stages (each representing one of the two specified polynomials) that are cascaded. Parallel implementations can be derived by analyzing the data flow through these serial shift registers.

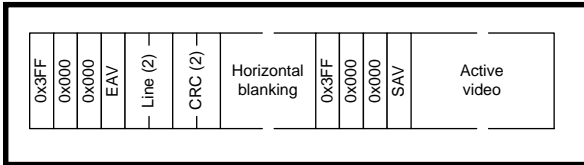


Figure 3—Digital video lines begin with the horizontal blanking interval rather than with active video. The TRS codes, line numbers, CRCs, and the rest of the blanking interval occupy the exact amount of time that is allowed for the electron guns in a CRT to return to the start of the next line. While SMPTE 292M sends the TRS, line number, and CRC information twice (once per chrominance and luminance channel), the aggregate time occupied by both channels must still conform to the CRT's blanking time.

platform must be capable of functioning with a 1000-ppm deviation or two different clock circuits need to be implemented and selected according to the desired mode.

292M ELECTRICAL BASICS

Both coaxial and fiber-optic physical layers are supported by 292M. Although cable length is not a standardized parameter, semiconductor vendors can implement high-performance receivers that allow real-world coaxial cable lengths of 100 m or more. Compare this to the 300 m that has become expected by the TV industry when dealing with the lower speed 270-Mbps 259M SD interface.

Fiber-optic cable permits greater distances, but you have to accept the added complexity and cost that high-bandwidth fiber optics brings with it. For in-studio applications, coaxial cable is an adequate and relatively inexpensive interconnect medium.

The 292M coaxial interface drives a 75-Ω cable that is terminated in its characteristic impedance. As in many communications circuits, it is desirable to minimize the DC level on the

coax, and hence necessary to minimize the DC content of the serial bitstream itself. You accomplish this by scrambling the data bits as they are transmitted by passing them through two separate polynomials:

$$G_1(x) = x^9 + x^4 + 1$$

$$G_2(x) = x + 1$$

These polynomials randomize the datastream and force frequent logic transitions on the wire. Regular data patterns containing long strings of ones and zeros are converted into seemingly random bits. When the same polynomials are applied in reverse to the received signal, the original regular data patterns emerge.

Figure 2 shows one possible shift-register implementation of the primary scrambling and descrambling step as illustrated in the 259M standard. As you see, the scrambling operation is simple when done in the serial data domain—just a series of shift registers with XOR gates placed at the bit positions specified by the polynomial. In the parallel domain, the logic circuit is slightly more complex but quite feasible.

292M DATA LINK STRUCTURE

Complete lines of video are composed of an HBI followed by an active region. The active region may or may not be visible onscreen, depending on whether the line is active or part of the vertical blanking interval (VBI).

Like the HBI, the VBI is dead time that gives the electron beam in the CRT time to return to the top of the screen to begin drawing the next video frame. Within the HBI portion of the 292M transmission are sample words that are reserved for synchronization and descriptive codes.

Figure 3 shows the organization of each line of video. The unused center portion of the HBI may be used to carry generic data—more on this later.

As shown in Figure 3, each line begins with an end-of-line (EAV) code that marks the beginning of the HBI. A corresponding start-of-line (SAV) code at the end of the HBI immediately precedes the active region. Collectively, these two basic types of codes are called timing reference signals (TRS).

The horizontal, vertical, and field sync pulses are encoded in the TRS codes and are extracted by the receiving equipment. Each TRS is marked by a three-word header (0x3FF, 0x000, 0x000) followed by the code word. The special values 0x000–0x003 and 0x3FC–0x3FF are excluded as valid video samples to allow their use as reserved signaling words.

This range is excluded to increase the compatibility of 292M with purely 8-bit processing systems. So, an 8-bit system won't see all ones or all zeros in the eight most significant bits of a sample.

The TRS code word contains the state of the three previously mentioned sync signals: H, V, and F along with error correction code (ECC) bits. As Figure 4 shows, the four ECC bits are the XOR of the three sync bits. This

Managing the Transition

Between now and the end of 2006, both DTV and legacy NTSC will be broadcast to our homes. Individual TV stations have the choice as to what programming will be broadcast in digital and analog. It is likely that a station's main programming will be simulcast on both media so that neither segment of the consumer market—those who adopt DTV earlier and those who wait until the last possible day—is left without coverage.

Advertising revenue, the lifeblood of a commercial television station, is based on a program's population coverage, so TV stations will ensure that they don't artificially reduce the numbers of potential viewers.

However, new business models and programming choices will emerge in the broadcasting industry as a result of the multichannel flexibility afforded by DTV.

Despite the new digital signal, people at home don't have to rush out and purchase expensive HDTV monitors. External receivers in TV set-top boxes let people view downconverted HDTV on their existing TV sets. PC plug-in cards enable others to watch HDTV on their computer screens and provide access to any broadcast data applications offered by a particular station. These consumer electronics products will allow a gradual adoption of HDTV at whatever pace an individual chooses.

ECC scheme allows for the implementation of a fairly robust and fault-tolerant point-to-point interface capable of correcting any single-bit error and detecting many multibit errors.

Following the TRS code is the current line number and a line CRC. The CRC is calculated over the previous line's active region, the following EAV code, and the line number. An 18-bit CRC value is calculated using:

$$\text{CRC}(x) = x^{18} + x^5 + x^4 + 1$$

The CRC is 18 bits long because only 9 of the 10 bits in each CRC word carry unique information. With the exception of TRS codes, link status samples such as line number and CRC have their most significant bits set to the logical inversion of the next most significant bit. This guarantees that one of the excluded values will never be forced into the datastream.

There's one more thing to know about the 292M data structure—all of the TRS codes and status words described above are sent twice per line! The 292M data link is divided into chroma and luma halves that are interleaved one sample at a time.

The result: separate TRS codes, line numbers, and CRCs for each chroma and luma channel. So, each CRC is calculated only for its channel.

If you're familiar with 259M—the SD interface—you wouldn't notice this just by looking at the active video pixels because they are interleaved chroma and luma just like in 259M. But when you observed the HBI, you'd see two 0x3FF samples followed by four 0x000 samples followed by two identical EAV code words.

It's worth mentioning here that one of the new video standards referred to is 480P. This 483-active-line progressive raster has 720 pixels per line.

Although the true terminology should be 483P, MPEG-2 requires the number of vertical lines to be evenly divisible by 16. Therefore, DTV broadcast considers only the bottom 480 lines. It is the same resolution as the digital version of NTSC (often referred to as 480I) but progressively scanned.

Although it's new, 480P isn't really HDTV. SMPTE 292M defines 480P at

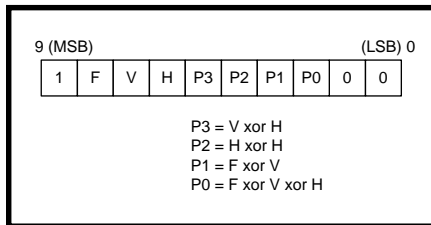


Figure 4—The built-in error detection and correction capability of TRS codes allows for sync recovery from low bit error rates and improves the integrity of the end-to-end data link. TRS correction circuits can be implemented with relatively few logic gates, making this feature practical and easy to build.

59.94 frames per second (fps) although there are some implementations at 29.97 Hz, making it extremely close to 480I in terms of quality.

There is currently no standardized method of transporting 480P over the 292M interface. This fact complicates studio cabling by requiring yet another set of wires and equipment just for this format. SMPTE is in the process of standardizing a mapping that will overlay the 480P raster on top of the 292M HD signal.

CHIPSETS FOR 292M

Now you know about the basic structure of the 292M data link, but you may wonder how one goes about handling a nearly 1.5-Gbps serial datastream. Two basic types of equipment operate on 292M: those that process the individual bits and those that simply pass through what was received.

The latter types are generally video routers that merely steer the incoming signal to one or more output ports. In that case, receivers, transmitters, and reclocking ICs are required. A receiver that can filter and amplify a 750-MHz data signal after it has traveled through 100 m of cable is not a trivial item.

Gennum, a video IC manufacturer based in Ontario, Canada, sells the GS1504 HD adaptive equalizing receiver as well as its companion transmitter, the GS1508 HD cable driver.

After traveling through 100 m of cable and being reconstructed by a receiver, a signal has had some amount of jitter introduced into it. Some edges are slightly shifted in time relative to others. At lower frequencies, this may not be a problem.

However, at edge-to-edge spacings of just over 1 ns, it doesn't take much

jitter to wipe out whatever timing margins are left in the system. Enter reclocking.

The reclocking process passes the serial datastream through a flip-flop such that the flip-flop's output is a clean version of the input with minimal jitter. The clock for this flip-flop is derived from the original datastream using a PLL. Gennum offers the GS1515 reclocker IC. AMCC, a well-known manufacturer of communications ICs, sells the S8301 HD reclocker.

But what if your product needs to process and manipulate the data contained within the 292M stream? It's impractical to expect to operate on the data serially at nearly 1.5 Gbps.

Each of the manufacturers I mentioned offers deserializer and serializer ICs that allow the data to be accessed in 20-bit-wide chunks at 74 MHz. AMCC has the S8501 HD deserializer and S8401 HD serializer, and Gennum has similar offerings—the GS1522 and GS1545, respectively.

These ICs provide access to the raw bits on the coaxial cable—scrambled, in other words. If your product contains sufficient additional logic resources, perhaps in an FPGA, to perform the scrambling and descrambling functions, either of these chipsets may suit your needs. Or if you need ICs that handle these low-level functions for you, you may want to consider Gennum's GS1501 and GS1500 companion ICs.

HD silicon, like the HDTV industry itself, is in a fairly early stage and there are always new developments. Be sure to check on the Internet for the most up-to-date offerings from these and other companies.

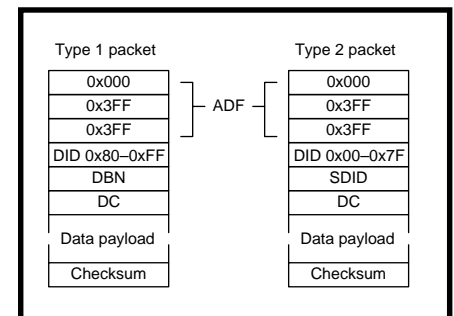


Figure 5—Ancillary data packets are carried in the blanking portions of the digital video signal that would be otherwise unused for carrying useful information. The packet format is relatively simple, enabling real-time parsing in hardware.

Group	299M DID
1	0x2E7
2	0x1E6
3	0x1E5
4	0x2E4

Table 2—Each audio group, consisting of two digitized stereo pairs (or four mono channels), is marked by a unique DID to identify it in the ancillary data space.

ANCILLARY DATA IN HD VIDEO

In considering the horizontal and vertical blanking portions of the SMPTE 292M datastream, you'll quickly notice the unused bandwidth and wonder whether it could be filled with useful data. Enter ancillary data!

From the days of SDTV and 259M, SMPTE created a standard dubbed 291M that sets forth an ancillary data packet structure that can be inserted into the video signal's blanking intervals. SMPTE is finalizing a similar standard for 292M that's based on the 291M packet structure.

Digital audio samples are the most common use for ancillary data, but there are many other applications. Any video-related datastream that benefits from being transported along with the video is a candidate for ancillary data. Closed-captioning information and video source/tracking IDs are two more examples.

Figure 5 illustrates the ancillary data packet structure. Note that there are two similar variations, denoted as Type 1 and Type 2.

Each packet begins with a three-word header (0x000, 0x3FF, 0x3FF) referred to as an ancillary data flag (ADF). An 8-bit data ID (DID) follows: only eight of the ten sample bits have unique DID information.

Bit eight of the DID contains even parity for the eight least significant bits. Bit nine, the most significant bit, is the logical inversion of bit eight.

DIDs between 0x00 and 0x7F signify Type 2 packets that contain a secondary DID (SDID) following the DID. DIDs between 0x80 and 0xFF signify Type 1 packets that contain a data block number (DBN) in

place of the SDID. The DBN forms a continuity counter for successive packets with a given DID.

An application may or may not take advantage of this field. A data count (DC) word follows that provides the number of data words in the packet's payload. The DC is an 8-bit quantity and therefore allows for a maximum of 255 payload words. A zero-length payload is legal.

The two most significant bits of the SDID/DBN and DC are treated in the same way as in the DID. The last word in a packet is a nine-bit checksum value that covers the nine least significant bits of every word in the packet from the DID to the last payload word. The ADF is not included in this calculation. The checksum word's most significant bit is the logical inversion of bit eight.

An ancillary data packet contains fairly low overhead and is easy to parse in both hardware and software. Given the dual-channel nature of 292M (chroma and luma), a single ancillary data packet must be properly formatted into either channel and may not straddle both at the same time.

Common ancillary data types have DIDs that are reserved by SMPTE for specific functions. Embedded digital audio, for example, has a total of eight DIDs assigned to it—four for SD and four for HD formats.

The HD ancillary data standard, still in process, reserves DIDs and SDIDs (Type 2 packets) for closed captioning and content advisory information. Additional DIDs are reserved for a variety of other functions.

Attempts were made to facilitate the processing of ancillary data packets by 8-bit-only systems—an 8-bit DID and DBN. However, this is not uniform because of the 9-bit checksum and the allowance for 10-bit data words in the payload. To fully process ancillary data, especially in the case of audio, a minimum 10-bit data path is required.

HD EMBEDDED AUDIO

It is highly desirable to embed digital audio samples within the digital video signal; every video program has accompanying audio. Doing so guarantees that a program's audio will not get lost or out of sync. It also simplifies the infrastructure of a studio by enabling a complete uncompressed program to be carried on one wire.

SMPTE defined the 272M standard for embedding audio in the 259M SD SDI signal. Likewise, SMPTE 299M addresses embedded audio for HDTV applications.

299M is a significant improvement over its SD predecessor, owing to its easy support of 24-bit samples and more coherent structure. Designing logic to handle 299M is easier than doing the same job in the SD world.

All audio samples are expressed as 24-bit quantities. If an ADC samples at a lower resolution, the smaller sample is placed into the most significant bits of the 24-bit data space and the least significant bit is zero-filled.

Audio sampling rates that are both synchronous and asynchronous with respect to the video clock are supported. The audio sampling rate is synchronous with the video clock if

the frequency relationship between the two can be expressed as a ratio of integers. Three synchronous rates are supported: 48, 44.1, and 32 kHz. The preferred mode of operation is 48-kHz synchronous sampling.

As in the SD context, 299M defines four groups of audio, each containing four channels. In most scenarios, the four channels are organized as two stereo pairs.

Bit	Sample Word			
	1	2	3	4
9 (MSB)	not bit 8			
8	even parity for each word, bits 0–7			
7	audio [3]	audio [11]	audio [19]	P ¹
6	audio [2]	audio [10]	audio [18]	C ¹
5	audio [1]	audio [9]	audio [17]	U ¹
4	audio [0] ²	audio [8]	audio [16]	V ¹
3	Z ¹	audio [7]	audio [15]	audio [23] ³
2	0	audio [6]	audio [14]	audio [22]
1	0	audio [5]	audio [13]	audio [21]
0 (LSB)	0	audio [4]	audio [12]	audio [20]

Table 3—A 24-bit resolution sample of a single audio channel spans four data words along with AES flag bits and parity information. Samples with lower resolution (e.g., 20 bits) are aligned with the most significant bit (bit 23), and the least significant bits are padded with zeros. ([1] AES flag bits; [2] LSB; [3] MSB).

Up to eight stereo pairs may be embedded within a single 292M videostream. The format for the audio samples is derived from the Audio Engineering Society (AES) audio-frame format that most professional digital audio is first put into after sampling.

Each complete audio group is assigned its own ancillary data packet with a reserved DID that indicates its group number. These packets are placed into the chroma channel of the 292M videostream. Table 2 lists the reserved DID for each HD audio group. Note that there are four separate DIDs reserved for the four SD audio groups.

The audio group data structure in the packet's payload is fixed length, regardless of the number of audio channels in use. Figure 6 depicts the format of a 299M audio group packet. This is a Type 1 ancillary data packet, so it contains a DBN that may optionally be used as a continuity indicator.

The DC is a fixed value, 0x218, indicating a 24-word payload length. Following the DC is a two-word clock-phase indicator that's used to regenerate the audio sampling clock for the group at the receiving end.

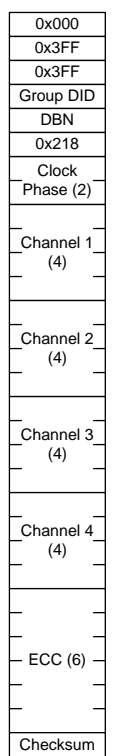
All channels in a group must be sampled with the same timebase, but different groups may use different timebases. This restriction is most useful in asynchronous sampling modes. In synchronous modes, the audio sampling clock may be obtained from the video block by means of a PLL if the sampling frequency is known beforehand.

Sample data for each of the four channels follows with each sample occupying four words. Table 3 shows how the 40 data bits are organized. The five flag bits (Z, P, C, U, V) are derived from the original AES stream.

Following the four channel samples are six ECC words that cover the words from the start of the ADF through the last sample. The standard ancillary data checksum is the last word in the packet. In the case of 299M, its utility is far surpassed by the coverage of the previous ECC words.

Parsing a 299M packet is easy due to its fixed length and structure. In applications that support only the common 48-kHz synchronous sampling rate, the clock-phase indicator can be ignored.

Figure 6—All four channels (generally, two stereo pairs) in a single audio group (packet) share the same sampling timebase. In situations where all four channels are not used, undefined data is substituted in their place because of the requirement to transmit a fixed-length packet containing four audio channels.



The 299M standard limits the number of audio data packets for a given group to a maximum of two in any horizontal ancillary data block. The regular sampling rate of an audio source, coupled with this burst restriction, enables you to specify a minimum and maximum receiver buffer size to ensure that once sample extraction begins, a valid audio stream never causes a buffer underflow or overflow.

Audio control packets are the second type of packets specified by 299M. These control packets are inserted into the luma channel of the 292M videostream, are sent once per video field, and are fixed length.

Each group has a control packet that provides information such as sampling rate, active channels within the group, delay of audio to video, and additional phase relations between audio and video frames. Depending on the application, it may be possible to ignore these control packets.

MPEG-2 AND BEYOND

In the process of broadcasting an HDTV program to your home, the SMPTE 292M standard takes an HD videostream from its origin, through studio editing equipment, and into an MPEG-2 encoder. Once inside the

encoder, it is compressed by a factor of roughly 80:1. It then exits as seemingly random bits. The MPEG-2 and ATSC data infrastructure that allows the compressed bits to be decoded and displayed is a huge topic that I'll discuss just briefly in Part 2.

DTV broadcast also brings with it concepts of multiple channels, video formats, data, and new business models, which I'll address in Part 3. ▣

Mark Balch is a senior hardware design engineer at DiviCom and has participated in a variety of MPEG-2 product designs, including an HDTV MPEG-2 encoder. Mark actively attends meetings of the ATSC and SMPTE industry standards groups. You may reach him at mark_balch@hotmail.com.

REFERENCES

- SMPTE 259M, *10-Bit 4:2:2 Component and 4f_{sc} Composite Digital Signals—Serial Digital Interface*, 1993.
- SMPTE 274M, *1920 × 1080 Scanning and Analog and Parallel Digital Interfaces for Multiple Picture Rates*, 1997.
- SMPTE 291M, *Ancillary Data Packet and Space Formatting*, 1996.
- SMPTE 292M, *Bit-Serial Digital Interface for High Definition Television Systems*, 1996.
- SMPTE 296M, *1290 × 720 Scanning, Analog and Digital Representation and Analog Interface*, 1997.
- SMPTE 299M, *24-Bit Digital Audio Format for HDTV Bit-Serial Interface*, 1996.
- www.atsc.org
- www.fcc.gov
- www.mpeg.org
- www.smpte.org

SOURCES

GS1504, GS1508, GS1515
 Gennum Corp.
 (905) 632-2996
 Fax: (905) 632-2055
 www.gennum.com

S8301, S8401, S8501
 AMCC
 (858) 450-9333
 Fax: (858) 450-9885
 www.amcc.com

Without Acceleration

FROM THE BENCH

Jeff Bachiochi

Part 1: All We Have Left is Velocity



Sensor technology now packages

sensors and signal conditioning in small surface-mount chips. Here's a way to design an accelerometer data-acquisition system using a sensor on a chip.



For many of us, acceleration and gravity bring to mind

Newton's laws of motion. And one can hardly think of Newton without the familiar tale of him getting bonked on the noggin with an apple. Because of the force of gravity, that famous apple went from hanging on the tree (not moving) to hitting the ground (and not moving) in a very predictable way.

Gravity applied constant force on the apple, causing it to fall. And at each instant in time, the apple's momentum was increased by this force.

The apple's change in momentum over time is acceleration. The force of gravity on the apple is defined as 1 *g*, which causes the apple to fall at an average velocity of about 9.8 m in the first second. If the tree was very tall and it took two seconds for the apple to fall, gravity would continue to increase the apple's momentum to 19.6 m/s in those 2 s. Gravity's constant *g*-force allows velocity to increase over time.

But gravity isn't the only means to acceleration. Any force acting on a mass can produce acceleration:

$$\text{acceleration} = \frac{\text{force}}{\text{mass}}$$

Because acceleration (due to gravity) is a constant (at our earth's radius), the force of gravity on a mass is proportional to its mass. Gravity has the ability to accelerate a mass, yet at any instant in time there is no acceleration, only velocity. Acceleration is a measurement over time:

$$\text{acceleration} = \frac{\Delta \text{velocity}}{\Delta \text{time}}$$

In the case of gravity, the acceleration equals 9.8 m/s per second. This is the distance any object falls (discounting other influences) toward the center of the earth, per second.

This measurement is called 1 *g* (where *g* stands for gravity). Acceleration is often described in units of *g*.

A dramatic illustration of acceleration is felt while driving a car. Every time we step on the accelerator, the engine speeds up, applying forces that increase the vehicle's speed. We feel these positive *g*-forces as we are pressed back into the seat.

At the point where we ease off the pedal and the velocity (speed) remains constant, the acceleration falls to zero. The *g*-forces that are still present prevent any change in velocity. When we apply the brakes, negative *g*-forces reduce the velocity. Then, we can feel our bodies rising out of the seat—a good reason to wear a seatbelt.

For a moment, let's revisit the

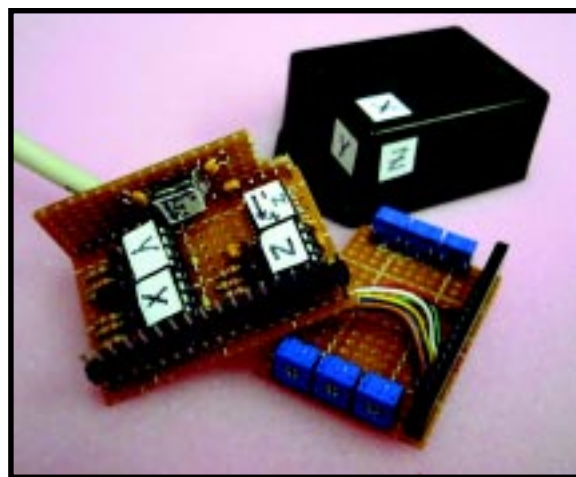


Figure 2—The ICs labeled x, y, and z are op-amps whereas the other two ICs (labeled with arrows) are accelerometers. x-y is the 250 and the z-axis IC is the 150. The board on the right holds six trim pots and mates with the sensor board to fit in the container at top right.

measurement of 1 *g*. If 1 *g* is a velocity 9.8m/s (or 32 ft./s), then let's see how fast this is in more familiar terms:

$$98 \text{ m/s} \times 60 \text{ s} \times \frac{60 \text{ min.}}{5280 \text{ ft./mi.}} = 21.8 \text{ mph}$$

In other words, if your vehicle could pull a constant 1 *g* of acceleration, it could do 0–60 mph in under 3s.

If we measure the amount of “seat-squish” due to acceleration, we could develop a scale for determining the *g*-force applied based on our body's movement. This is how a semiconductor accelerometer works.

ADXL150/250

Analog Devices has been fabricating sensor and signal-processing circuitry on the same chip since 1993. Like the micromachined motors seen on science programs some time now, the sensor element is created by depositing polysilicon on an oxide layer.

When the oxide layer is dissolved, the polysilicon is suspended in midair (so to speak). You can imagine how delicate this sensor is. A drop to the floor may be enough to ruin it. It's not the fall that does it, but the sudden stop.

Remember, the acceleration (or deceleration) is:

$$\frac{\Delta \text{velocity}}{\Delta \text{time}}$$

Going from some speed to 0 instantaneously is a very large number. These devices have a maximum 2000-*g*-force rating unpowered and maximum 500-*g*-force rating while powered.

The sensor looks like multiplate variable capacitors, with 42 capacitor cells forming the sensing structure. Each cell has fixed plates and a moving plate connected to a moving beam.

The linear beam moves in one axis, like your body squishing into the car's seat. Linear motion produces a differential change in capacitance which is measured by the on-chip circuitry.

The ADXL150 holds a single *x*-axis sensor, while the ADXL250 has *x* and *y*-axis sensors mounted at right angles to each other.

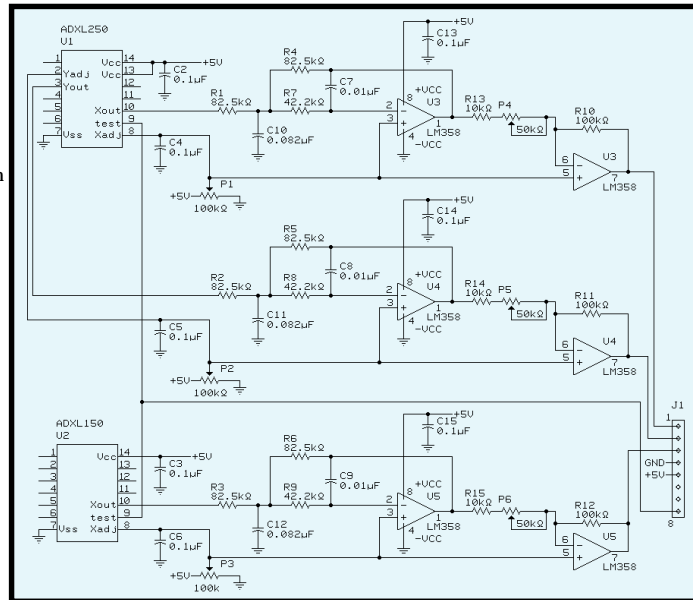


Figure 1—Two sensors provide three axes of output. Each axis has its own signal conditioner consisting of a filter and amplifier.

One of the great additions to this sensor is an array of 12 electrostatic cells that can force the movement of the beam. This self-test feature is enabled through an input pin. Applying a logic high to the input pin allows both the mechanical and electrical systems to be checked for proper operation.

The onboard clock, demodulator, filter, and buffer amp are all that you will need to convert the capacitive change to a voltage output. Both the sensitivity and 0-*g* values are ratio-metric to the supply voltage for this sensor. Thus, using the sensor's supply voltage as the A/D reference makes everyone happy.

The voltage output of the sensor for 0 *g* is set at 0.5 V_{CC} , so both negative and positive forces can be measured. Sensitivity of the device is ~38 mV/*g*.

Even though today's 5-V rail-to-rail op-amps can approach outputs of ground and V_{CC} , it is still a good idea to steer clear of the rails whenever possible. A swing of 2 V above and below the 0-*g* output (2.5 V) will indicate ±52.6 *gs*:

$$\frac{2V}{0.038 \text{ V/g}} = 52.6 \text{ g}$$

WHEN 50 IS TOO MUCH

To monitor this sensor I want to use a PIC processor. The '16C63 pro-

cessor has internal an 8-bit ADC. Based on a 5- V_{CC} , each bit of resolution would represent 19.5 mV:

$$\frac{5V}{256 \text{ bits}} = 0.0195 \text{ V/bit}$$

At 38 mV/*g* output from the sensor, that's 0.5 *g*/bit of A/D resolution. Depending on what you're looking for, this may or may not be a reasonable quantity.

In some instances we might be interested in recording down to 0.1-*g* increments or less. If so, we'd need to add some gain (a factor of 5) between the sensor and the ADC. Here's

where signal conditioning becomes important.

Using a single op-amp in an inverting amplifier configuration:

$$\text{Gain} = \frac{R_f}{R_{in}}$$

I'm going to design for a minimum gain of 2 and a maximum gain of 10 (practical).

A series 10-kΩ fixed and a 50-kΩ pot as the R_{input} resistance with a 100-kΩ $R_{feedback}$ resistance will furnish gains of 10 (100k/10k) and 1.6 (100k/60k) at the two extremes of the pot. Since eight-pin IC DIPs offer dual as well as single op-amps in the same package, the second op-amp can add additional filtering (see Figure 1).

The sensor's onboard filtering limits the bandwidth to 1 kHz. Even limited to 1 kHz, the device noise can be considerable if you're interested in sub-*g* measurements. Peak-to-peak noise can be as high as 0.4 *g*. That's more than ±3 bits, if your circuit gain is 5 (to give a 0.1-*g* resolution).

By limiting the bandwidth down to 100 Hz, you can reduce the peak-to-peak noise by a factor of ~4, or less than ±2 bits. Not only does the gain of the circuit increase the sensor output and noise, but also any DC offset from the nominal 0.5 V_{CC} is increased as well.

Without gain, the bias offset can be as much as ±10 *g*—by far the biggest culprit to overall errors. The sensor's

offset null input allows this offset to be set to zero via a trim pot or fixed resistor to V_{CC} or ground.

ADJUSTING THE ADJUSTMENTS

Gravity can be used for the offset and gain adjustments. When the sensor's measurement axis is placed parallel to the surface of the earth, gravity has no effect on the sensor.

The offset adjustment can be trimmed to produce one-half V_{CC} (2.5 V) at the output of the final op-amp stage. This is the 0-g output state.

Rotating the sensor's measurement axis perpendicular to the surface of the earth will place 1 g on the sensor. Depending on the polarity of the axis, the sensor output will go up or down.

You can adjust the gain pot to guide the sensor's output to the appropriate voltage. Choose this voltage based on what you want for a full-scale output. For instance, if you want a 5-g full scale output—that is, the voltage excursion from 0 to 5 g is 2.5 V (2.5 V to 0.5 V)—then each g will output 0.5 V.

That's it. Set the output to 3 V (2.5 V + 0.5 V), or if the axis is reversed, 2 V (2.5 V - 0.5 V). Don't forget to go back and readjust both pots again if necessary. After final adjustments, you should see the circuit's output go between 2 and 3 V as you rotate the sensor from on-axis through off-axis and back on the reverse-axis.

A TO D TO ASCII

To record data, we need to digitize the analog output and save it to a file. The analog output from the accelerometer is connected to a PIC16C73 (Figure 1), which has an onboard ADC.

I want the processor to sample the ADC and output the conversion value at 19,200 bps. The value will be output as a three-digit decimal number.

I want to do conversions on three channels. So, the ASCII output will be of the format `xxx-yyy-zzz<cr><lf>`. These 13 characters will take just over 6 ms. But I'm forced down to a sampling rate of less than 200 S/s, which isn't so bad, considering that the

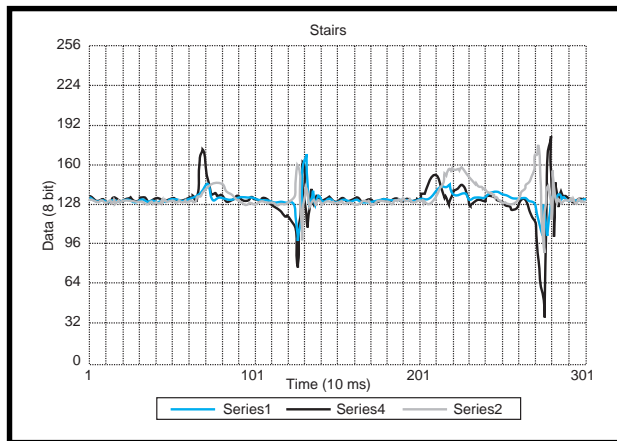


Figure 2—This graph shows two steps of my right foot. The positive excursions are the lift and forward forces while the negative excursions are the sudden stops as my foot hits the floor.

bandwidth of the sensor is 1000 Hz and I've filtered it down to 100 Hz to improve the noise floor. So, the final design is three channels at 100 Hz each.

A laptop PC can display the three channel values onscreen because the data is formatted with `<cr><lf>`. And, the datastream can be recorded to a file for later display.

I wrote the PIC software a bit differently than I usually do. Everything is done in three interrupt routines: a timer overflow, an A/D conversion complete, and a transmit buffer empty.

The 3.579-MHz resonator has a period of 279 ns (1/3,579,500). Execution time is 1.117 μ s (279 ns \times 4), making timer0 with no prescale overflow every 286 μ s (1.117 \times 256).

Every time that timer0 overflows, the overflow interrupt routine reduces a counter (initialized to 35) and then exits the interrupt. Nothing happens in the main loop.

Eventually, the timer0 counter is reduced to zero in the interrupt routine. Instead of leaving the routine, the counter is set back to 35 and an A/D conversion is started on the first channel. At this point, 10 ms has passed (286 μ s \times 35).

Next, an A/D conversion-complete interrupt occurs. In the conversion-complete interrupt routine, the conversion value is stored into the transmission buffer. The channel is incremented, a new conversion is started, and the interrupt routine is exited.

Again, back to the main loop where nothing happens (except for periodic

timer0 overflow.) When a conversion of the final channel is complete, the first channel is selected but no conversion is started (new conversions don't happen until timer0 overflows 35 times, remember?) This time the transmitter empty interrupt is enabled.

On exiting from the last A/D interrupt, a new interrupt immediately takes over. The transmit empty routine loads a character into the SBUF register for transmission out of the UART. This rela-

tively quick interrupt routine exits back to the main do-nothing loop.

The UART is double-buffered, which means that the next character can be loaded while the previous character is still being transmitted. This feature ensures that the characters will be transmitted one right after another. Because the transmit buffer is of a known length, once all the characters are sent, the transmit empty interrupt is disabled.

The linear buffer is preloaded with the space and `<cr>` and `<lf>` characters. This only needs to be done once because the A/D converted values are stuffed into the buffer at the same place each time, unlike a ring buffer in which all the data must always be completely written. Now, execution remains in the main loop until the timer0 again overflows 35 times and the A/D conversion begins again.

FIRST DATA

It's time for the first test of this system. The outputs are in a `xxx-yyy-zzz` format. However, this format assumes that the module can be fastened such that the orientation remains the same as the module's markings. That won't always be so. Should the module have to be fastened differently, then the `x-y-z` outputs must be translated such that the data matches that of the correct axis.

For this test, the sensor (shown in Photo 1) is strapped to my right foot. The umbilical cord connects to a laptop's serial port which I will carry so that I can be mobile, uh...sort of.

Let's take a ride on the elevator at *Circuit Cellar* World Headquarters. I start the laptop, and using HyperTerminal, I click on Transfer and Capture Text. The output data scrolls on the screen enroute to the hard drive.

A brief walk down the hallway leads to the elevator. I shuffle in and hit the lobby button. Whoosh. The hydraulics are released and I drop in a regulated free fall.

At the lobby, the doors part and I press the 2 button to return back to the second floor. Now, the pump has to work to force the elevator back up. This time, the ride is much slower because gravity can't be used. Exiting the elevator at my floor, I again click on Transfer, Capture Text, and Stop to complete the test.

The datafile is simply a list of numbers from all of the A/D conversions. It doesn't look too interesting or readable. Fortunately, there are tools available to make this data easier to comprehend.

I used Excel's spreadsheet for this project. Importing data is simple if

you understand how Excel interprets the file. Most importing algorithms look for a delimiter to separate the data into columns. Delimiters are spaces, commas, semicolons, and so on. Each row of data ends in a <cr>.

If you play by these exact rules, the imported data is placed in a column/row format directly into the spreadsheet's grid. Here's where all the fun begins.

Graphing commands let you choose any or all of the spreadsheet data and plot it in a number of different ways. But wait just a minute—something must be wrong here. The graph shows all of my steps, but where is the vertical movement of the elevator?

Hmm, the floor-to-floor height is ~16'. The time to move between floors is 8 s. That's 2 ft./s. Oh, that's only a small fraction of a *g*. So, let's look at only a small portion of the graph. Figure 2 is just a portion of the total test data. This part of the graph (walking) is much more interesting than the vertical lift.

THE END?

As you can see from this project, using an accelerometer isn't difficult. Although using three sensors is probably overkill for most applications, designing in an accelerometer for vibration analysis, crash sensing, active suspension, and a slew of other applications may make perfect sense.

But, I'm not done yet. There are a few changes I want to make to this project that will greatly improve it. Here's a hint. I want to eliminate two things: screwdrivers and wire. ☒

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.

SOURCE

ADXL150/ADXL250

Analog Devices
(617) 329-4700
Fax: (617) 329-1241
www.analog.com

SILICON UPDATE

Tom Cantrell

LPC

The Little Processor that Could



LPC may stand for “low pin count” or “low price

chip” or any number of things, but this new family of '51s from Philips offers much more than fun with words. Tom has all the info on this new microcontroller.



good 20 years after it was born, the 8051 remains one of the most popular 8-bit MCUs. In fact, according to the market stats I've seen, the '51 solidly occupies second place, behind Motorola's 68HC-05 and well ahead of Microchip's PIC.

If this seems surprising, it's probably because unlike the Motorola and PIC architectures, which are essentially sole-sourced, the '51 is practically a people's micro. It's offered by dozens of companies as the basis for standard chips, specialized variants, and ASIC cores.

More surprises when you zoom in on the '51 roster. Who's the biggest slugger? Is it Intel? Nah, they've got bigger chips to fry. How about Atmel? No, despite the great things they've done with flash '51s. Siemens? Oki? Dallas? Nice try, but wrong again.

The answer is that the IC division of that European consumer electronics giant, Philips Semiconductors, is number one in '51s.

Frankly, I wouldn't be surprised if you have little idea of who Philips is or any clue as to how they ended up controlling a huge chunk of the MCU biz. All the more surprising, considering the iffy track record of offshore suppliers (European or Japanese) penetrating the tough U.S. MCU market.

As a supplier of TVs and stereos and such, Philips has only recently started

selling under their own name in the U.S., rather than under various labels (Magnavox being one of the most familiar). “Aha,” you say, “they may ship a lot of '51s but most are buried in their own consumer gadgets.”

Wrong again. Although there is a lot of internal use, the fact remains that the merchant-market '51 you buy is more likely to have the Philips shield on it than any other logo.

How did Philips manage to pull it off? The answer is that they got a running start when they acquired Signetics, one of the influential Silicon Valley semi shops. I realize some of you may never have heard of Signetics since, as with Magnavox, the name was brought under the Philips banner years ago.

The point is that the acquisition bought Philips a key advantage because “Sig” (as locals called them) already had a viable '51 business going, built on a long-ago official licensing deal with Intel. This meant that Philips could take advantage of the existing product line, distribution channels, customer base, and perhaps most importantly, a stable of local, experienced talent.

Between the combination of the original Philips and Signetics '51 lines and the subsequent addition of new variants, the Philips '51 family now includes a dizzying array of parts that pretty much covers the spectrum.

I say “pretty much” because Philips arguably hasn't kept up with the pack when it comes to addressing the surging demand for very small and low-cost chips, a trend that Motorola and Microchip have jumped all over.

LOW PIN COUNT

That is until now. Enter the new Philips LPC (low pin count) family of '51s. The first part, the LPC764, shown in Figure 1, includes 4-KB OTP EPROM and 128 bytes RAM, and comes in a 20-pin package (both DIP and surface mount). According to Philips, subsequent products will work their way down with less memory (a forthcoming LPC762 has 2-KB OTP EPROM) and smaller packages, with 16-pin and even 8-pin versions on the horizon.

LPC also means “low price chip.” The '764 is rolling out at \$1.21, and

that only requires a PO for 500 parts, which suggests that below-buck pricing is in reach for high-volume customers.

At first glance under the hood, there isn't much to distinguish the '764 from the other '51 variants on the market, or for that matter, the circa-70s original. However, although the basic configuration and peripherals appear familiar, there are a number of enhancements that, taken as a whole, significantly update and freshen the design.

Those improvements start with the '51 core itself, which we can see from Figure 1 has been accelerated. It cuts the number of clocks required to execute each instruction in half (which in most cases is from 12 to 6).

Running at up to 20 MHz puts performance around 3 MIPS, which is what's expected of an entry-level MCU these days. Effectively, it allows the '51 architecture to maintain its place in the performance standings as competitors make similar speed improvements.

Those of you who've upgraded an existing design with an enhanced CPU know that timing differences related to CPU and peripheral operations can be a hassle, in the worst case requiring all the hardware and software to be reviewed and retuned.

By contrast, CPU and peripheral timing is simply twice as fast for the '764 as for a standard '51. There's even an OTP configuration bit that divides the clock in half for set-and-forget '51 timing compatibility.

This timing discussion leads us to the clock oscillator, shown in Figure 2, an area significantly upgraded from the traditional '51. There are a total of five clock source options including

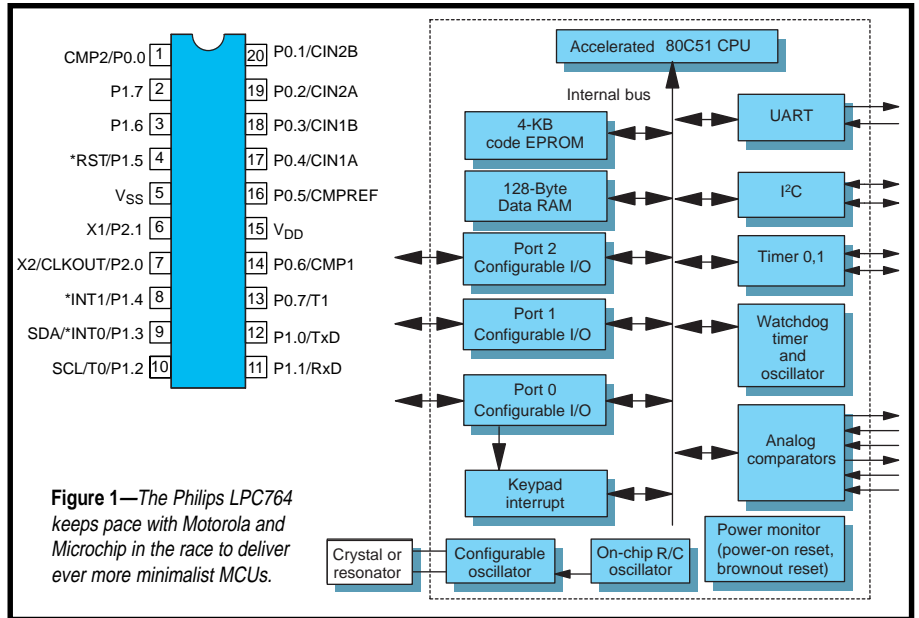


Figure 1—The Philips LPC764 keeps pace with Motorola and Microchip in the race to deliver ever more minimalist MCUs.

external, internal, and three flavors of crystal: 20–100 kHz, 100 kHz–4 MHz, and 4–20 MHz (the latter two modes support operation with a ceramic resonator, as well).

The internal oscillator is especially welcome, saving the cost and board space of a crystal and its requisite RCs, and boosting reliability (one less thing to break). It runs at 6 MHz, but loose $\pm 25\%$ accuracy rules it out for timing-critical applications.

Note that there's a control bit that allows the internal clock (divided by one sixth, i.e., 1 MHz) to be optionally output to a pin (X2/CLKOUT/P.2.0) for synchronous designs. Otherwise, the pin is available for use as I/O. Generating and keeping the clock on-chip has the benefit of reducing EMI. That goal is further achieved with low slew rate (10-ns minimum rise/fall times) output drivers on the I/O lines.

Whatever the clock source, the '764 runs it through an 8-bit divider with divide ratio equals $2 \times (n + 1)$ for $n = 1-255$ (i.e., cutting the clock rate between a factor of 4 and 512 before sending it on to the rest of the chip.) Unlike the divide-by-1-or-2 timing compatibility feature which is set with an OTP configuration bit, the 8-bit divider is dynamically programmable in normal operation (i.e., anytime; not just during reset, etc.).

The chip takes care of making a smooth switchover to the new clock so you don't have to be concerned about timing glitches or disrupting anything—an automatic transmission, if you will.

LOW-POWER CONTROLLER

The programmable clock and other power-saving features make the LPC well suited for battery-driven apps. Supply voltage range is a wide 2.7–6 V, but note that speed is restricted at lower voltages (i.e., less than 10 MHz for V_{DD} less than 4.5 V).

Even running full-bore, the LPC doesn't use a lot of power—typically 15 mA at 5 V (20 MHz) and only 4 mA at 3 V (10 MHz). Note that there's a control bit, LPEP (low-power EPROM), that you can set if the supply voltage is less than 4 V. This bit disables on-chip circuits only required for higher voltage, cutting power consumption even further.

Two low-power modes stretch the battery budget even further. Idle mode

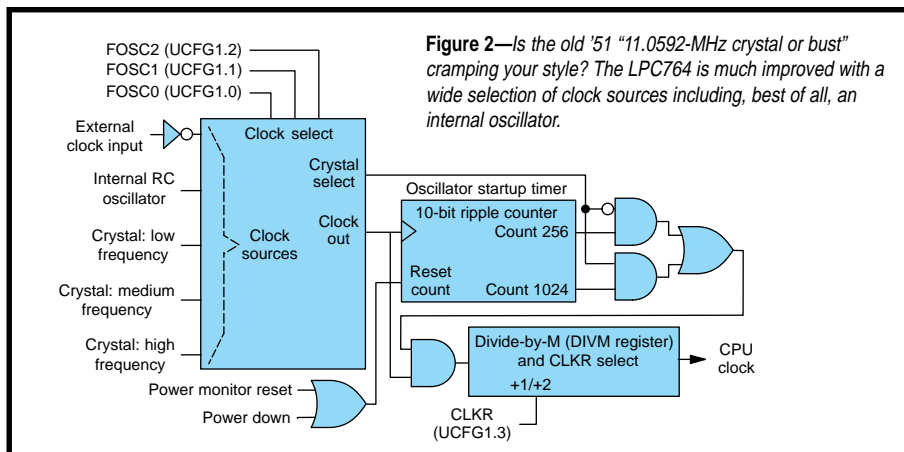


Figure 2—Is the old '51 "11.0592-MHz crystal or bust" cramping your style? The LPC764 is much improved with a wide selection of clock sources including, best of all, an internal oscillator.

only cuts power consumption roughly in half but leaves the oscillator and all peripherals running for quick wakeup. By contrast, power-down mode shuts practically everything down, cutting consumption to mere microamps.

However, waking up from power-down is a bit sluggish. You have to wait while the oscillator starts up (256 clocks for the internal oscillator, 1024 clocks for external crystals). One improvement is that wakeup from powerdown, traditionally requiring a reset, is now possible using an (enabled) interrupt source.

In powerdown, the supply can be cut to 1.5 V and still retain the RAM contents. But note that the SFRs (special function registers; i.e., peripherals, etc.) aren't guaranteed below the 2.7-V operating minimum. You can either refresh those that matter in your wakeup software or take advantage of the new software reset, which initializes the SFRs (but doesn't touch RAM).

The most significant power-management upgrade is the addition of brown-out detection with selectable trip voltages of either 3.8 or 2.5 V. The default response to brownout is to reset the processor, but optionally an interrupt can be generated instead.

Your boot code can take advantage of a brown-out flag (BOF) and power-on flag (POF) to figure out just what caused the latest reset. Do note the fine print concerning power supply rise/fall timing required to guarantee proper operation of the brown-out feature.

LOTS OF PERIPHERALS CRAMMED

If you take advantage of the on-chip clock and reset, a full 18 of the '764's 20 pins are available for I/O, and the chip manages to do quite a bit with them, even within the constraints of '51 compatibility. The traditional functions (timer/counters, UART, parallel I/O, etc.) are all there, but with a variety of useful upgrades.

For instance, the two 16-bit counter/timers, running at up to one-sixth the clock rate (i.e., greater than

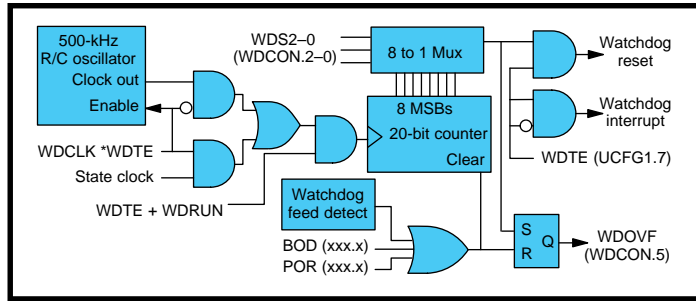


Figure 3—The watchdog timer features its own independent oscillator, eight different timeouts, and selectable reset or interrupt on overflow.

3 MHz for a 20 MHz '764), feature the usual modes. However, an added pin toggle-on-overflow feature is especially useful for PWM and other waveform generation tasks.

There's also a watchdog timer that runs off either the CPU clock or its own 500-kHz oscillator (see Figure 3). It offers eight timeout options between 8k and 1M clocks (~16 ms to 2.1 s).

Using the watchdog's own oscillator increases reliability because it forces the chip into reset should the CPU clock fail. Make sure to consider the fact that the watchdog oscillator is only $\pm 37\%$ accurate when choosing a timeout value.

The watchdog function is enabled or disabled by an OTP configuration bit. If the function is not required, the watchdog can serve as a simple interval timer, with status bit and/or interrupt overflow detection.

Like the timer, the UART features all the traditional modes and such, including the now widely used ninth data bit mode. This is especially useful for multidrop networks in which the ninth bit differentiates between an address and data byte. When originally devised, the idea was that an address

byte would interrupt each node, but only the node with that address would have to deal with subsequent data bytes, leaving the others free to do something useful.

That notion is a pretty good one, but considering the small packets likely encountered in simple apps, it's a hassle that each node has to check every address. So, the 'LPC764 extends the ninth data bit concept with automatic address recognition (i.e., now a node is only interrupted with an address byte that matches its own ID).

Using two SFRs—an address (SADDR) and a mask (SADEN)—allows concocting clever group and broadcast addressing schemes. For instance, a three-node system could be mapped such that each node is independently addressable, each pair of nodes is also addressable, or all can be addressed at once.

This system reduces node overhead to the absolute minimum because a node is only interrupted by stuff it needs to see. Also, this helps to ease synchronization of activities across multiple nodes since a message can be sent to all of them at the same time.

Another popular serial interface the '764 provides is I²C. Since Philips invented I²C, it's not surprising that this is far more than the tattered-up shift-register impersonator found on some chips.

Those of you who've spent any time dinking with I²C know that simple bit-banging is OK for connecting a low-speed chip or two, but getting the high-speed and multimaster modes working right is nontrivial. To that end, the '764 I²C subsystem includes hardware features like bus

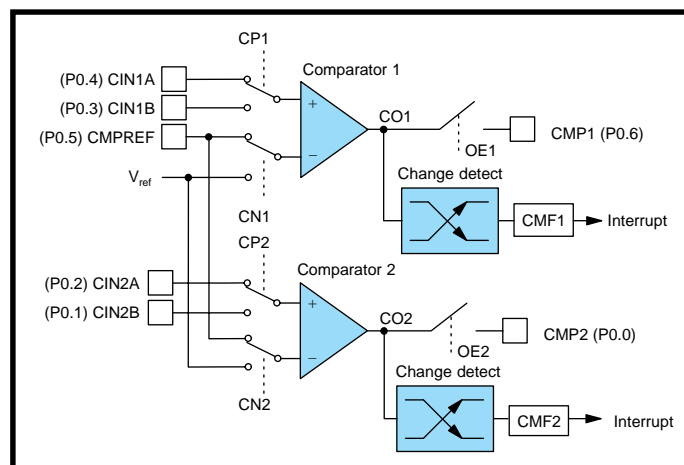


Figure 4—The dual analog comparators, with programmable pin assignment, internal or external reference, and pin and interrupt outputs, are quite versatile.



Photo 1—Exploiting a Philips bond-out chip (makes internal signals accessible), the EB-764 emulator board delivers high-end features (like real-time trace) yet doesn't cost an arm and a leg.

arbitration, bus timeout, and clock stretching that handle the gory details, so your software doesn't have to.

On the analog front, there are two comparators that can be configured in a variety of ways (see Figure 4). To sum up, the output of the comparator, which can be monitored via status bit or routed to a pin, is 1 when the positive input (one of two selectable pins) is greater than the negative input (a pin or an internal 1.28 V $\pm 10\%$ reference). Optionally, an interrupt can be generated every time the comparator output changes state.

The comparators continue to work in idle and power-down modes. The good news is, the comparator can serve as the alarm clock (interrupt) to wake the chip up. The comparator's pin output will continue to work, but switching time may be slower (unless the pin is configured in push-pull mode).

However, the comparator consumes power, so if you don't need it, make sure to set the disable bit. It takes $\sim 10 \mu\text{s}$ to stabilize if and when it's reenabled.

Even the parallel I/O lines have been upgraded with selectable bit-by-bit configuration (push-pull, open-collector, quasi-bidirectional, and input-only) and optional port-by-port Schmitt triggers with hysteresis on inputs.

Each pin can drive up to 20 mA, plenty for LEDs and such, though as usual the combined total output of all pins is limited by package thermal considerations. A keyboard interrupt allows easy detection of activity on any or all pins of port 0.

LET'S PROGRAM CHIPS

One hallmark of the '51 family is lots of tool support, and the '764 is no exception. Thanks to the maturity of the architecture, the selection, quality, and robustness of available tools is good.

For ASM, C, and such you can choose from dozens of suppliers—just download an LPC764-specific header file from the Philips web site that defines chip-specific registers and you're off to the races. Because the '764 is '51 compatible, you can also use an existing '51 emulator or SBC for preliminary development and prototyping. Although it won't know about the '764 upgrades, it's possible to either ignore or work around the differences early on.

Philips offers an '764-specific emulator for \$399 and also resells Ceibo's EB-764, shown in Photo 1, for \$299. That's a good deal for a full-featured unit that, exploiting an authentic Philips bond-out chip, offers full-speed emulation and real-time trace.

There are plenty of shareware tools to get you started as well. The software that comes with the EB-764 can be downloaded from Ceibo's web site and includes a Windows development environment with, notably, '764-specifics built in (see Photo 2). Meta-link also has a demo version of their popular Windows package, though it only targets a generic '51.

Eventually you'll end up with a .hex file that has to be burned into the '764's



Photo 2—You can download the software that comes with the EB-764, including the LPC764-specific simulator, from Ceibo's web site.

OTP EPROM. The chip is programmed using a clock serial scheme requiring five pins: clock, data, Vp-p (10.75 V), 5 V, and ground.

It's a matter of issuing a simple command sequence (i.e., set address, write, start and stop programming, read) to program and verify the OTP. Programming time is only 250 μ s per byte, which adds up to just a second or so for the entire chip.

The '764 is suitable for in-system programming on the production line. Feel free to add a small header for blow and go as long as you avoid pin conflicts (e.g., the pin used for Vp-p [10.75 V] also serves as the external reset input).

If you take advantage of the on-chip reset, no problem, but make sure that whatever's connected to the pin can take 10.75 V without choking. Watch out for possible gotcha specs such as Vp-p minimum rise/fall time (1 μ s) and minimum V_{DD} to Vp-p delay (20 μ s) that rule out hot-plugging the chip.

Although hacking your own programmer is fun, those in a hurry should probably just pick up the \$99 Philips

programmer (also resold by Ceibo). It connects to a PC serial port and includes built-in Vp-p generation. Also, many popular party programmers (BP, EETools, Needhams, etc.) offer upgraded drivers that support the '764.

Philips is one of the many MCU suppliers that have hopped on the emWare bandwagon, so if you want to hang a '764 on the Internet, check out the Link-51 evaluation kit. Although it's an emWare-specific (not general-purpose) development tool, the kit does come with the Metalink assembler and the price is definitely right at \$79.95.

LE PERFECTO CONTESTO

If you think the '764 is right up your alley, the Design2K contest offers you the perfect chance to check it out. So, all you '51 gurus out there, now's the chance to show those sissies who think you need a 32-bit RISC to run a toaster what embedded is all about. ☒

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for more

than ten years. You may reach him by e-mail at tom.cantrell@circuitcellar.com, by telephone at (510) 657-0264, or by fax at (510) 657-5441.

SOURCES

51LPC764

Philips Semiconductors
(408) 991-5207
Fax: (408) 991-3773
www.semiconductors.philips.com

EB-764

Ceibo
(314) 830-4084
Fax: (314) 830-4083
www.ceibo.com

EMIT

emWare
(801) 256-3883
Fax: (801) 256-9267
www.emware.com

Assembler

MetaLink Corp.
(480) 926-0797
Fax: (480) 926-1198
www.metaice.com

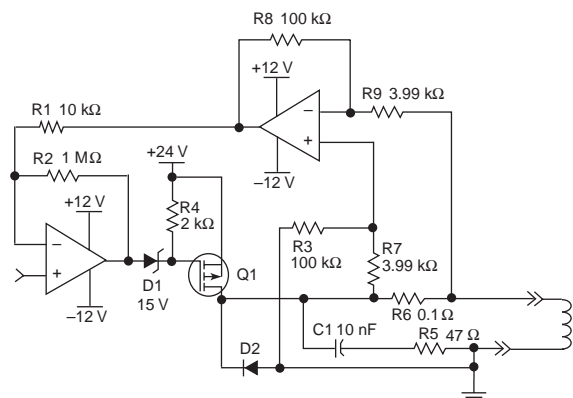
CIRCUIT CELLAR Test Your EQ

Problem 1—Can you write a function in C that will return a number indicating the bit position of the most significant bit in a byte? Can you write the function without using a compare statement such as “if”? The function should return 0 if no bits are set, and 1 indicates the least significant bit is the most significant bit set. A sample function prototype is given below:

```
int ms_bitpos(unsigned char b);
```

Problem 2—You are asked to mix an 8.5-Molal solution of NaCl in water for calibrating a new water-activity instrument you are developing. You obtain a box of reagent-grade NaCl, because run-of-the-mill table salt has way too many impurities. You fill up a pitcher with deionized water. Describe the procedure for mixing the calibration solution. Describe the difference between Molarity and Molality.

Problem 3—The circuit shown (proportional current driver) is a free-running PWM current driver for ground-referenced loads. The output current is proportional to the drive voltage. What is the function of the zener diode D1?



Problem 4—You are designing electrical equipment for use in a glovebox that's used for sorting low-level radioactive waste like clothing, tools, and supplies. You want to use really good wire for this job and you know that Teflon-insulated wire is great stuff with extraordinary temperature range. Would this be an acceptable choice?

PRIORITY INTERRUPT

Another Typical Trip



t

his year's Embedded Systems Conference in San Jose illustrated that embedded controls are still a booming business. There were more 32-bit processor and tool vendors than you could shake a stick at. It's a bit disconcerting for hands-on engineers like myself to admit that designing and implementing microcontrols has become 10% hardware and 90% software. Design finesse is still a revered goal in engineering departments, but inexpensive embedded PCs lead many designers to simply nuke the problem with processing power and clean up the fallout in software. I'm learning to accept it.

ESC exemplifies the trend in this industry—the big get bigger and the small can't afford to display at this show. Although it's nice to visit the Intel and Microsoft traveling circuses, I prefer to meet the entrepreneurs and engineers who are designing products that will become tomorrow's Microsoft or 3-Com. Fortunately, the Embedded Internet Workshop is still new enough to have the distinct flavor of discovery and entrepreneurship.

As a pure business decision, I continually evaluate the merits of attending these shows. After all, sending a half dozen people to a show for a week not only impacts the schedule but gets damn expensive. But, I get to press the flesh with readers and I come away with a pocket full of author contacts.

That's the good news. The bad news is that I don't like traveling and this last trip didn't help the prospects for the next one. Don't get me wrong. I'm not a hermit. I'll drive 400 miles to spend a nice weekend somewhere. But, tell me I have to fly cross country and live out of a suitcase for a week and it's nothing but dread. Why? Because stuff always happens. This last trip was no exception.

Normally my wife and I fly out of Hartford (BDL) on United Airlines. This time there was almost a \$1000 difference in cost between tickets from BDL and Providence (PVD) so we stayed overnight at PVD and left at 6:15 AM on American for San Jose (SJC), with a plane change at JFK. The rest of the staff flew out of BDL.

I should have known things were going too well. Both shows were great and I was anxious to get home so I could act on all the commitments I had made. We got to the airport 1½ hours before the return flight to JFK. A half hour before takeoff they changed the gate. Then they said there was a delay due to equipment problems. One more gate change and 2½ hours later, we lifted off while the captain made excuses. Five hours later and 1½ hours late, we landed at JFK.

We saw the puddle-jumper that we were supposed to take to PVD boarding as we taxied in. We vaulted (as much as someone my size is capable of doing) out of the plane and ran to our connection. The door was closed and the best we could get from any airline personnel was, "I don't know." After watching the plane to PVD taxi out, we went back to the airline desk and were informed that we could wait 6 hours, go across town to Laguardia, and fly out on US Air. Forget the luggage. Put in a claim and it might show up in a couple days.

Eventually we found out that airlines typically dump all the luggage from missed connections. We walked through the whole terminal looking for the right dumping ground. At the last carousel we saw our suitcases in a heap. Mine looked odd. The locks had been cut off, the web belt I secure around it was missing, and the contents had been tossed.

At that point, renting a car and driving 175 miles from JFK to Providence seemed far more appealing than subjecting myself to more airline abuse. Unfortunately, we'd have to drive 60 miles past our house to get to Providence, only to pick up the car and drive back to Connecticut, but our choices were limited. After 5 hours, 3 construction sites, and 2 accidents on I-95, we pulled into Providence.

I pulled up to our car to dump the luggage before returning the rental. What? Dead battery?! In my haste to toss something in the car when we left a week ago, I must have left the dome light on. Well, what is AAA good for if not to call at 1 AM to start your car? He didn't even snicker.

At 3 AM we pulled into our driveway. Everything seemed normal. I punched in the code. The HCS and the alarm system did their thing and we entered the house. I picked up the pile of unread mail and clicked the answering machine at the same time. As I read a note from the Resident State Trooper to call him about "damage to the wall," I heard, "Hi, this is Genevieve next door. If you had your driveway video system on last week you may want to check it. There's an article on page 18 in Wednesday's *Journal* about a high-speed car chase and crash into someone's driveway. I think it was your driveway!" Beep.

I quickly dug through the pile of newspapers. Car chase? Criminals? The biggest crime in this town in the last 5 years was a rash of smashed mail boxes. We don't even have a police force. Eventually I found the article. Some guy had robbed a liquor store and stolen an SUV in the next town. Eight police cars chased him through Vernon into a long residential driveway. Guess whose.

I grabbed a flashlight and went outside. It didn't take long to notice that he had crashed through a free-standing railroad-tie planter that even my big diesel tractor with the backhoe couldn't hope to move. I walked back to the house and shook my head.

After the events of the day all I could say was, "Well, it was another typical trip to the West Coast! I can't wait for the next one."

steve.ciarcia@circuitcellar.com

A handwritten signature in black ink, appearing to read "Steve Ciarcia".