EMBEDDED PC MONTHLY SECTION

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#92 MARCH 1998

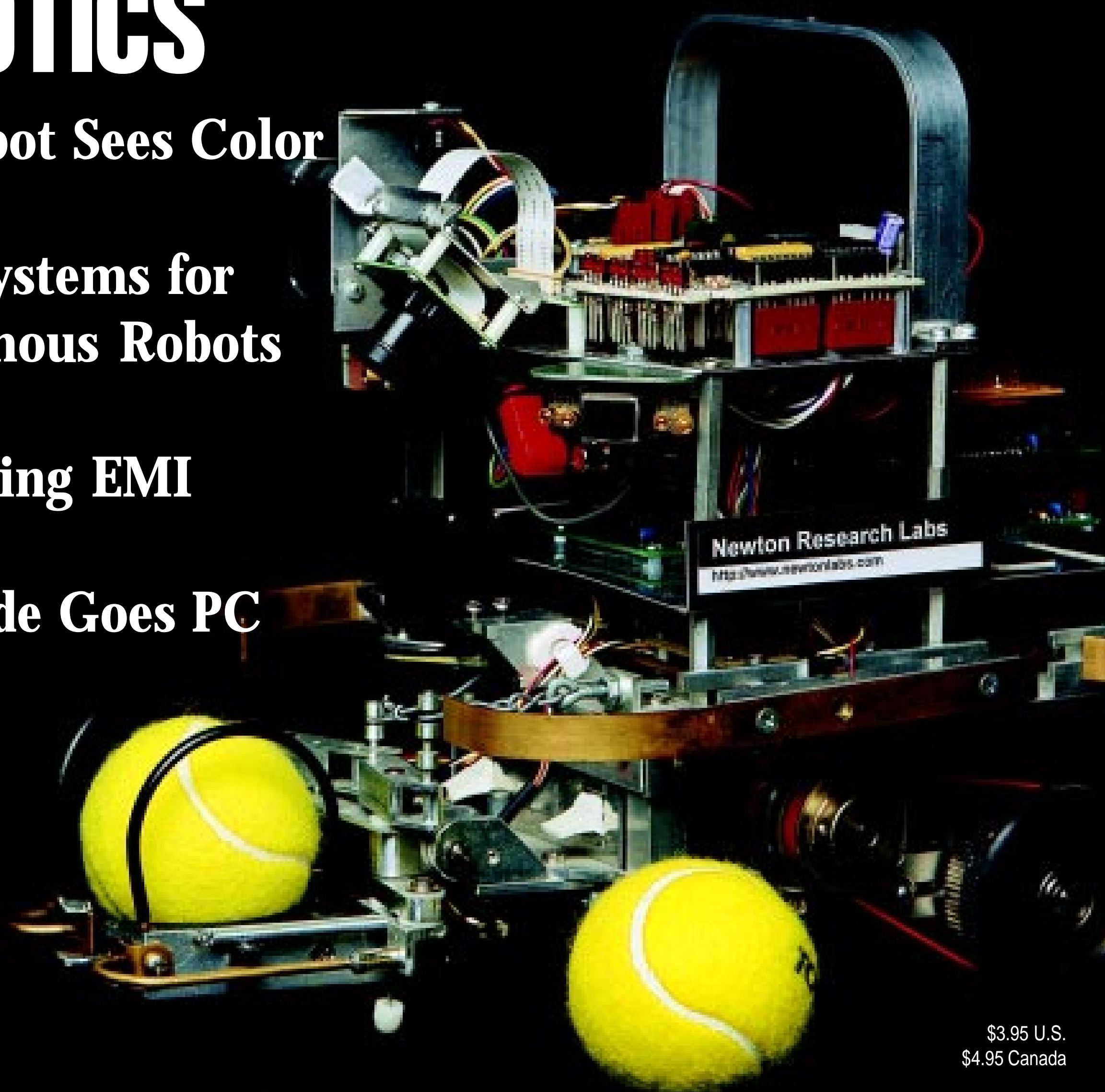
ROBOTICS

This Robot Sees Color

Power Systems for Autonomous Robots

Suppressing EMI

8051 Code Goes PC



$3.95 U.S.
$4.95 Canada

TASK MANAGER

Reality Alert

When robots first entered our pop culture mentality, it was with great fanfare and hype. People imagined critters who could come and take over all the sloth of their lives and dispense with nasty tasks at extraordinary speed. Car oil would be changed, bathroom sinks unclogged, lawns cut and watered, sidewalks shoveled—all with simple voice commands. After all, these jobs are very basic and every human being from two on knows how to speak, at least, sort of.

The fall from this illusion was as catastrophic for some as Adam and Eve's expulsion from the Garden of Eden. Now, any mention of robotics is met with scornful contempt. All robots have entered the never-never land of Star Trek, Star Wars, and Johnny 5. It's just not a reality.

Meanwhile, however, robotics marches on in many university and corporate laboratories. Frankenstein-like engineers piece together software and hardware technology with knowledge from oceanography, linguistics, mechanical engineering, neurology, physiology, anthropology, and so on. It's rather an endless list.

And the breakthroughs by these scientists are phenomenal. Robots that can leave a mothership for several hours to map a section of the ocean floor before returning to the ship. Prosthetic robots for people with severe spinal injuries. With these, a headset on the user transmits commands to a nearby slave prosthetic, thereby enabling them to be more self-sufficient. All-terrain wheelchairs granting someone access to the same places as able-bodied people. All of these applications require a blending of expertise from many disciplines.

Our first feature (and star of our front cover) is a good example of this same kind of discovery. Newton Labs, a pioneer in robot engineering, introduces us to M1, a color-sensitive robot. While M1's primary task of chasing tennis balls seems rather limited, the technology behind what the robot accomplishes is being used for autonomous spacecraft docking, automated acquisition of cargo by helicopter, and inspection of products ranging from fruit to upholstery.

Ingo Cyliax, who has worked extensively with the miniature Stiquito robots, brings us the next feature—how to power autonomous robots. Bruce Reynolds takes us back to the basics. His MicroBot is just plain fun. His goal: to help a non-techie friend discover how to program Intel's 8749 and learn the fundamentals of sequential control logic, servo control, timing, and so on. Gordon Dick wraps up the features by zeroing in on microprocessor control of motor speed, a necessary evil in many robot applications.

In *Embedded PC*, Chip Freitag and Jeff Kirk help you port your 8051 code to the embedded-PC world. Ingo illustrates how to pick a PC RTOS using a robot application, and Fred goes embedded via the PC Card. As Fred points out, a lot of functionality and client-specific tailoring can be accomplished by implementing PCMCIA technology.

In Part 2 of his series on designing for EMI, Joe DiBartolomeo reviews suppression components. Jeff shows how to do a workaround using software when traditional hardware UARTs won't do. And, Tom introduces us to Patriot Scientific's ShBoom CPU, a micro that incorporates some hot ideas from the past with cutting-edge developments of the present.

Janice

janice.hughes@circuitcellar.com

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

EDITOR-IN-CHIEF
Ken Davidson

MANAGING EDITOR
Janice Hughes

TECHNICAL EDITOR
Elizabeth Laurençot

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Rick Lehrbaum
Fred Eady

NEW PRODUCTS EDITOR
Harv Weiner

ASSOCIATE PUBLISHER
Sue (Hodge) Skolnick

CIRCULATION MANAGER
Rose Mansella

BUSINESS MANAGER
Jeannette Walters

ART DIRECTOR
KC Zienka

ENGINEERING STAFF
Jeff Bachiochi

PRODUCTION STAFF
John Gorsky
James Soussounis

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

ADVERTISING

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

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

CONTACTING CIRCUIT CELLAR INK

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

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

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

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

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

Circuit Cellar INK® makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors. Furthermore, because of possible variation in the quality and condition of materials and workmanship of reader-assembled projects, *Circuit Cellar INK®* disclaims any responsiblity for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in *Circuit Cellar INK®*.
Entire contents copyright © 1998 by Circuit Cellar Incorporated. All rights reserved. *Circuit Cellar INK* is a registered trademark of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

INSIDE ISSUE 92

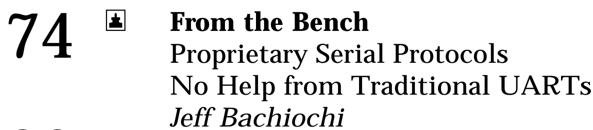
12 **Robots with a Vision**
Using the Cognachrome Vision System
Bill Bailey, Jon Reese, Randy Sargent, Carl Witty, & Anne Wright

24 **Power Systems in Autonomous Robots**
Ingo Cyliax

30 **MicroBot**
Programming Intel's 8749 for Robotic Control
Bruce Reynolds

58 **Motor Speed Control with a Microtwist**
Gordon Dick

66 **MicroSeries**
EMI Gone Technical
Part 2: Suppression Components
Joe DiBartolomeo



74 **From the Bench**
Proprietary Serial Protocols
No Help from Traditional UARTs
Jeff Bachiochi

80 **Silicon Update**
ShBoom Box
Tom Cantrell

Task Manager 2
Janice Hughes
Reality Alert

Reader I/O 6

New Product News 8
edited by Harv Weiner

Advertiser's Index 65

Priority Interrupt 96
Steve Ciarcia
For Once, I Sort of Agree

EMBEDDED PC

36 **Nouveau PC**
edited by Harv Weiner

41 **Converting 8051 Code for an 'x86 Embedded Processor**
Chip Freitag & Jeff Kirk

47 RPC **Real-Time PC**
Picking a PC RTOS
Ingo Cyliax

53 APC **Applied PCs**
Embedding PC Card
Part 1: The Time Has Come
Fred Eady

www.circuitcellar.com

READER I/O

BAD LIC. PL. # = CE 4 PDA OS

Edward Steinfeld's article was an excellent overview of why *not* to use Windows CE ("Windows CE is Ready, But for What?" *INK* 88). After being frustrated by Microsoft's previous orphaned attempts at making a lightweight OS (Windows Lite and Windows for Pens), I've been skeptical about this new offering. While I could point out a couple showstoppers for my application right away, it's good to have such a compilation of the OS's weaknesses before finding out the hard way. I still don't think Microsoft is serious about this product or the field in general.

A surprise: given the choices out there, it looks like the OS for scalable applications may be Linux. On the high end, Linux benchmarks significantly faster than most commercial OSs. It's also much more reliable and supported than anything I've used before.

For the embedded market, Linux can be booted from 1 MB of ROM and provides the capabilities of a mature multitasking, multi-user, and network-centric environment. Also, it supports a wider range of the '*x*86 platforms than WinCE. Full source means no problems with forced obsolescence or orphaning by the vendor. Drivers and tools are plentiful, and commercial contract help is available when custom work is necessary.

Development and deployment are trivial. It's the same OS scaled across the platforms as opposed to a retrofitted or redesigned version of a desktop OS. And for those areas where commercial support is crucial, there are many competing companies from which to choose—not just one unresponsive monopoly.

Thad Starner
testarne@media.mit.edu

MEET THE PIONEERS

The keyboard diagrams and table found in Table 1 of "A Hardware Keyboard Remapper" (*INK* 89) were built from research done by Altek Instruments. Altek's work represents a significant accomplishment as virtually all previously published information they found was in error in some way. If you're interested in learning more about building a keyboard wedge interface, be sure to check out <www.hello. co.uk/altek/mule.html>.

Many thanks to Lee Allen for his permission to use the figures.

Cheng-Yang Tan
cytan@fnal.gov

GETTING HOTTER AND HOTTER

We appreciated Fred Eady's article, "Interfaces and GUI-Building Packages—Part 2: Emulating Paper Tape" (*INK* 89), which used emWare's embedded networking tools to create an interface to a paper-tape reader. The article was interesting, informative, and accurate at the time it was written.

In late September 1997, we released V.2.0 of the EMIT software. This version addresses the limitations mentioned in the article by incorporating RS-485 multidrop protocol, dial-up modem support, and Internet Explorer and Netscape compatibility. Also, the Netscape plug-in is no longer required, and we added drag-and-drop user interface programming with Symantec's Visual Café.

Todd Rytting
www.emware.com

ADVANCED NEEDS TO BE EXPENSIVE—*NOT!*

I found "Building Advanced Device Drivers for the MPC860" (*INK* 90) interesting, but lacking in one area. Avi forgot to mention that DriveWay for the MPC860 costs about $30,000! Aisys's Web site lists prices from $500 to $1200 for versions of DriveWay for various 8- and 16-bit processors. But, you have to call to get a quote for the '860. I nearly choked when I found out its cost was an order of magnitude higher!

The MPC860 version is an impressive tool, but it's priced too high. It may be a small chunk of change at GM, but $30,000 represents about five years of the engineering software budget at my company. I'll just have to keep looking for $2–3k C compilers and as many shareware libraries as I can find.

Would the Aisys license keep me from becoming a driver-generating consultant? If I could sell off the source code, perhaps I could make a living writing drivers for the '860—or maybe someone is already doing it cheaper than Aisys. They charge about $5k to run front-end specs through their program! Probably takes 5–10 minutes to process your specs and generate the code. Pretty expensive minutes.

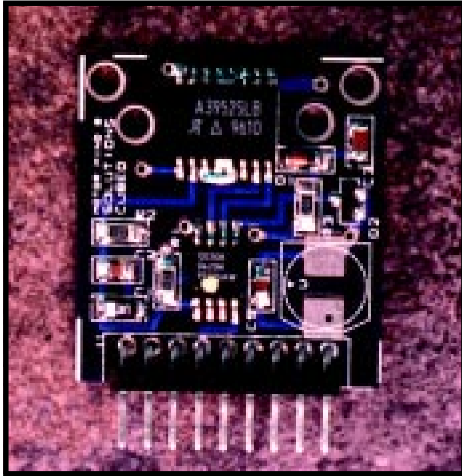
Mark Borgerson
borgerm@peak.org

CORRECTION

ChorusOS URL (*INK* 90, p. 60): www.sun.com/chorusos

NEW PRODUCT NEWS

Edited by Harv Weiner



MOTOR MIND B

Motor Mind B is a serial DC motor rriver module controlled by commands sent through a one- or two-wire interface. Its short instruction set enables the user to implement complex control algorithms quickly and with little effort. Bidirectional or unidirectional DC motors with operating voltages up to 30 VDC, peak currents as large as 3.5 A, and continuous currents of 2 A can be handled. Package power dissipation must not be exceeded during use.

Features include the ability to read a motor's tachometer frequency (0–65,528 Hz), automated speed control, 254 discrete steps of speed control, and motor direction changes. A watchdog timer eliminates the possibility of a system firmware failure. The Motor Mind B comes in a 1.2″ × 1.3″ SIP module. Its small size and connection scheme enable the device to be inserted directly into circuit boards for production runs or into breadboards for easy prototyping.

The Motor Mind B sells for **$29.95** in single quantities. It can be purchased directly from Solutions Cubed and is distributed by Parallax, Jameco, Marlin P. Jones, and Digi-Key. Complete datasheets and application notes are available via the Solutions Cubed Web site.

Solutions Cubed
3029 Esplanade Ste. F
Chico, CA 95973
(530) 891-8045 • Fax: (530) 891-1643
lon@solutions-cubed.com
www.solutions-cubed.com **#501**

HIGH-SPEED IR CONTROLLER AND TRANSCEIVER

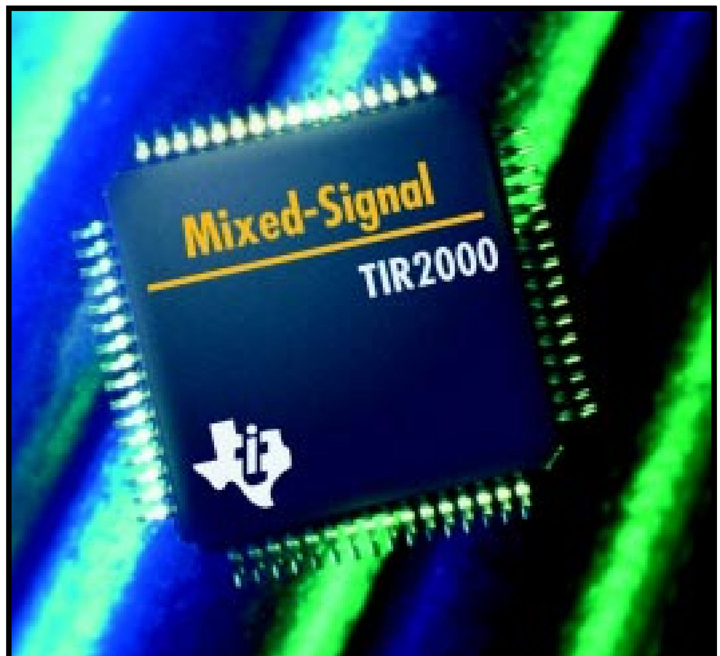
A new infrared controller and transceiver that provides a fully compliant high-speed IrDA solution is available from Texas Instruments. These devices support IrDA, the main standard for IR data communications, up to 4 Mbps, as well as amplitude shift keying (ASK) and television IR standards on the controller. The IR controller and transceiver are ideal in applications such as PC and notebook computers, printers, PDAs, and telephones.

The IR controller, designated the **TIR2000**, is an interface between the ISA bus and an IR transceiver that encodes and decodes information so that it conforms to the appropriate standard and can be understood and communicated by multiple systems. The TIR2000 also converts the data into a format that can be transmitted by the IR transceiver. The TIR2000 has a smaller pin count than any other 4-Mbps IrDA solution currently on the market, which saves board space.

The IR transceiver, designated the **TSLM1100**, includes a PIN photodiode, a two-path receiver with LED driver, and an 870-nm LED. The TSLM1100 interfaces directly with an IrDA controller and operates at data rates from 2400 bps to 4 Mbps.

The TIR2000 and TSLM1100 are available from TI and its authorized distributors. The TIR2000 is available in a 64-pin TQFP with a suggested price of **$6.42** in quantities of 1000. The TSLM1100 has a suggested price of **$5.55** in quantities of 1000.

Texas Instruments, Inc.
Semiconductor Group, SC-97072 • Literature Response Ctr.
P.O. Box 172228 • Denver, CO 80217
(303) 294-3747
www.ti.com/sc/5052 • www.ti.com/sc/5800 **#502**



NEW PRODUCT NEWS

DATA ACQUISITION STARTER KIT

The **DI-150RS Starter Kit** is a low-cost solution for two-channel data-acquisition and waveform analysis using a PC serial port. A user can digitize and store a transducer's analog output with 12-bit accuracy at rates up to 240 samples per second. At the same time, the transducer's output can be viewed onscreen in a triggered sweep or scrolling display format.

The DI-150RS is equipped with two analog input channels that can be software configured as two single-ended channels or one differential channel, both with a gain of 1 or 100. It includes a thermistor input and regulated excitation output and derives its power directly from the RS-232 serial port line.

WinDaq software provides data acquisition, real-time display, disk streaming, and playback and analysis of the acquired signals. It enables review and analysis of waveforms with smooth scrolling in either time direction as well as any degree of waveform compression. Data files can be imported and exported from a variety of data-acquisition, spreadsheet, and analysis formats.

The software's disk-streaming design enables data files of any length to be graphically displayed and browsed. Seven standard cursor-based time and amplitude measurements, frequency domain (FFT and DFT), and ten statistical analysis functions simplify waveform analysis and interpretation. Digital filtering permits graphical editing of the power spectrum for high-pass, low-pass, band-pass, and notch filters.

The kit features the DI-150RS module, serial communications cable, two-channel version of WinDaq data-acquisition software, WinDaq Waveform Browser software for playback and analysis, and documentation. In short, everything needed to acquire and playback data is available for **$99.95** (two-channel unit).

Dataq Instruments, Inc.
150 Springside Dr., Ste. B220
Akron, OH 44333-2473
(330) 668-1444
Fax: (330) 666-5434
www.dataq.com

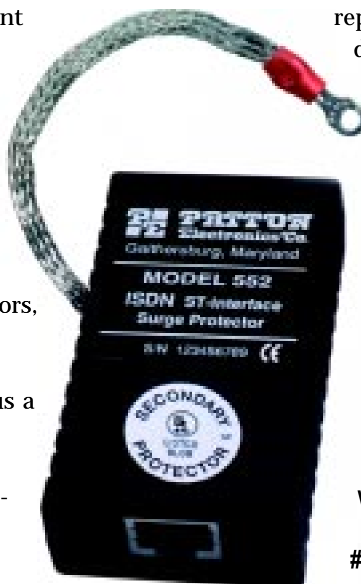
 # 503

TELEPHONE LINE PROTECTOR

The Patton **Model 552 Series** secondary surge protector contains seven different versions for protecting T1, E1, PRI, ISDN/U, ISDN/ST, DDS, two-wire dial-up, and 2-/4-wire leased-line telecom circuits. Installed between an incoming telecom line and a modem, CSU/DSU, or similar device, the Model 552 guards sensitive hardware against damage from nearby lightning strikes, electric motors, and other sources of transient surges.

The Model 552 is equipped with modular (RJ-11 or RJ-45) I/O jacks plus a sturdy metal braided strap. Transient energy is intercepted before it causes hardware damage, and is safely diverted to nearby chassis ground through the strap. The Model 552 is UL 497A listed for secondary surge protection and can handle repeated surges up to 1500 W. Its "fail safe" design feature causes the protector to fail short to ground in the event of a catastrophic surge, thereby sacrificing the protector to save connected equipment.

Prices range from **$39 to $89** per unit, depending on the type of interface and the number of pins protected. A 6″ (15.24 cm) patch cable is included with each protector.



Patton Electronics Co.
7622 Rickenbacker Dr.
Gaithersburg, MD 20879
(301) 975-1000
Fax: (301) 869-9293
www.patton.com

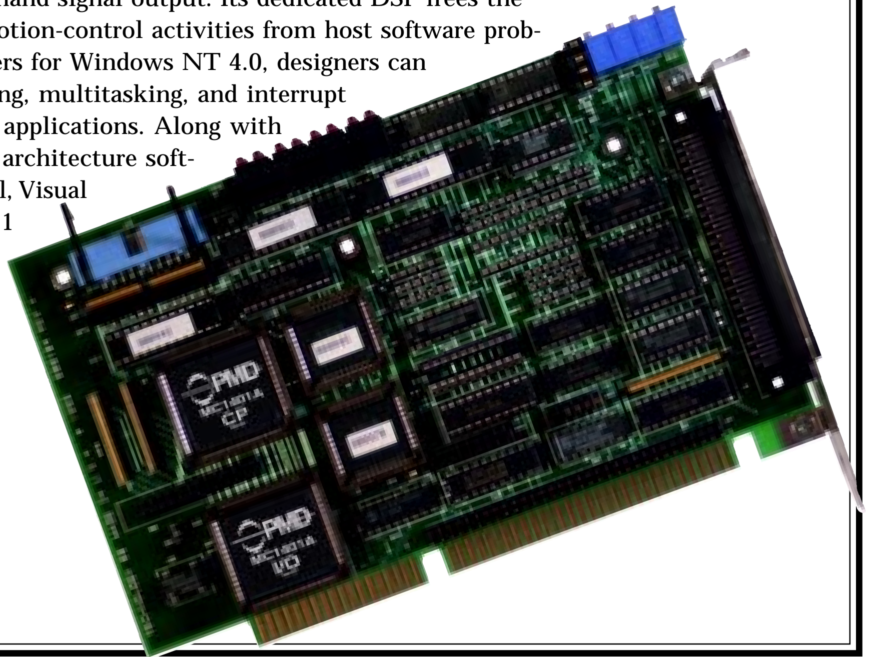
#504

NEW PRODUCT NEWS

DSP SERVO CONTROLLER FEATURES WINDOWS NT SUPPORT

The **Model 5650A** is an affordable, board-level servo controller that offers S-curve, trapezoidal, and velocity motion profiling; 31-bit position, velocity, and jerk registers; as well as 16-bit DAC or 10-bit PWM command signal output. Its dedicated DSP frees the host CPU for other tasks and protects motion-control activities from host software problems. With the introduction of new drivers for Windows NT 4.0, designers can take full advantage of NT's multithreading, multitasking, and interrupt capabilities for PC-based motion-control applications. Along with the NT drivers, the Model 5650A's open architecture software library supports C, C++, BASIC, Pascal, Visual Basic, and 16-bit drivers for Windows 3.11 and Windows 95.

The Model 5650A PC servo controller sells for **$950**.



Technology 80, Inc.
658 Mendelssohn Ave. N
Minneapolis, MN 55427
(612) 542-9545
Fax: (612) 542-9785
info@tech80.com
www.tech80.com **#505**

FEATURES

12 Robots with a Vision

24 Power Systems in Autonomous Robots

30 MicroBot

58 Motor Speed Control with a Microtwist

Robots with a Vision

Using the Cognachrome Vision System

FEATURE ARTICLE

Bill Bailey, Jon Reese, Randy Sargent, Carl Witty, & Anne Wright

Having a problem mastering Sampras's serve? This robot, equipped with a color-conscious vision system, can chase down your wayward tennis balls. You'll find it's a grand slam system. Game, set, and match to M1.

machine vision has been a challenge for AI researchers for decades. Many tasks that are simple for humans can only be accomplished by computers in carefully controlled laboratory environments, if at all.

Still, robotics is benefiting today from some simple vision strategies that are achievable with commercially available systems.

In this article, we fill you in on some of the technical details of the Cognachrome vision system and show its application to a challenging and exciting task—the 1996 International AAAI Mobile Robot Competition in Portland, Oregon.

MACHINE VISION

Vision systems typically have the architecture depicted in Figure 1a. But, this way of processing images has a problem. There's too much data in the video streams.

The NTSC video standard (used in North America, Japan, and several other parts of the world) provides about 240 lines of video at 60 frames per second. It takes a very fast CPU to do any significant processing at that rate.

The sorts of CPUs typically used in embedded systems generally can't

process video at the full 60 Hz. Ranges from 1 to 5 Hz are much more common.

The Cognachrome solves the problem by using hardware acceleration to do relatively simple vision processing (see Figure 1b). This strategy improves performance while reducing overall cost. The Cognachrome simplifies the vision task by looking for only three colors, which the system is trained to see.

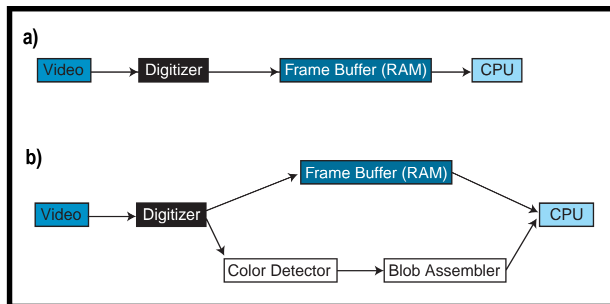


Figure 1a—*A typical machine vision system loads digitized pixels into RAM for the CPU to process in software.* ***b—****The Cognachrome system achieves 60-Hz tracking with special hardware that detects pixels of interest and assembles them into contiguous blobs.*

During operation, the acceleration hardware compares each pixel against these colors and groups contiguous pixels of the same color into abstractions called "blobs." Client software then uses the location and size of blobs (as well as other information about them) to identify and react to its environment.

The Cognachrome uses a simple fixed-coordinate system to refer to the video image. The horizontal axis ranges from 10 to 230 (left to right), and the vertical axis ranges from about 10 to 240 (top to bottom.) The exact numbers depend on the particular camera used.

Photos 1a and b demonstrate how the Cognachrome processes a video image. The hardware presents the Cognachrome with the blobs as shown in Photo 1b. For each blob, the Cognachrome computes several interesting statistics, including:

- the *x* and *y* coordinates of the centroid (i.e., the center of gravity)
- the area (number of pixels)
- the bounding box (the *x* coordinates of the left- and rightmost pixels and the *y* coordinates of the topmost and bottommost pixels)
- the aspect ratio (how elongated the blob is). A value of 3 indicates that the blob is three times as long as it is wide.
- the orientation (only meaningful for elongated blobs; the direction of the long part of the blob)

The user can add other useful statistics.

The Cognachrome can compute these statistics at frame rates up to 60 Hz. The actual frame rate achieved depends on the number of blobs in the image, their sizes, and which statistics are computed. Aspect ratio and orientation are much more expensive to compute than centroid, area, and bounding box.

When not requesting aspect ratio and orientation, the system can handle 10–20 blobs quickly. If aspect ratio and orientation are included, it may only handle 5–7. If the system is overloaded with too many blobs, it drops to 30 Hz or less.

The Cognachrome can be used to grab frames into a frame buffer and do more traditional vision processing on them. Resolution is lower in this mode (e.g., 64 × 48 to 64 × 250). The frame rate in this mode depends on the vision processing being done, but software-only processing is unlikely to be better than 30 Hz, even for the simplest processing.

USING THE VISION SYSTEM

The Cognachrome can either be used as the main processor in an embedded system or as a peripheral to another computer.

In embedded use, the user programs the vision system by registering a callback function. The callback is invoked after every frame of video and has access to all of the blob data. Statistics are not computed for a blob until requested, avoiding unnecessary computation.

COGNACHROME HARDWARE

The vision board has NTSC video input and output jacks, which provides a lot of flexibility in the choice of cameras. We put small CCD camera boards on our mobile robots. Size isn't as much of an issue for stationary applications, so we often use camcorders for their flexibility and low cost.

The Cognachrome has a video output jack for viewing the blobs in real-time black and white, which is useful during color training. The video comes from the color-adaptive recognition phase of the hardware.

Many of the hardware resources of the 68332 are available for embedded applications, including digital I/O lines, several TPU (timer coprocessor) lines, a bus with software-definable chip selects, and one synchronous and two asynchronous serial ports.

THE CONTEST

Every year, the AAAI (American Association for Artificial Intelligence) holds robot competitions at its annual

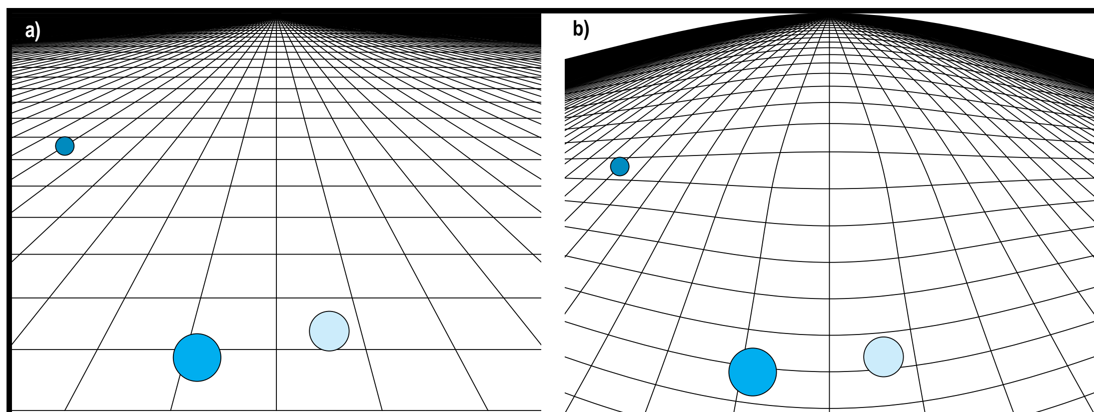


Figure 2a—*With an ideal video camera, M1 would see the world much like this.* ***b—****The world, as seen through M1's actual camera, has fish-eye distortion, which is typical of cameras that show a wide field of view.*

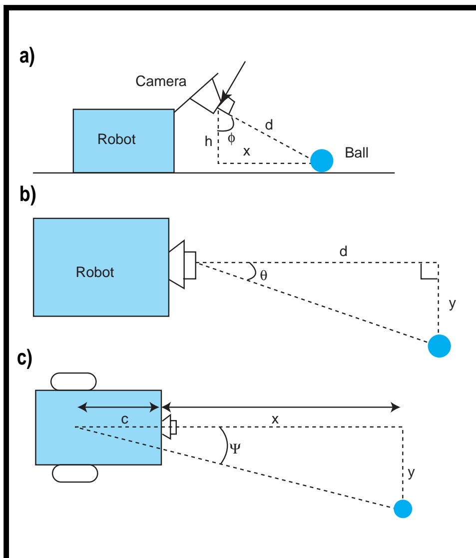


Figure 3a–c—*M1 can determine a ball's distance and angle relative to the robot from the ball's location in the camera image, assuming the ball touches the floor.*

conference. In 1996, the contest was for an autonomous robot to collect 10 tennis balls and 2 quickly and randomly moving, self-powered squiggle balls and deliver them to a holding pen within 15 min.

At the time of the conference, we had already been manufacturing the Cognachrome for a while and saw this contest as an excellent way to put our ideas (and our board) to the test. We outfitted a general-purpose robot called M1 with a Cognachrome and a gripper and wrote software for it to catch and carry tennis balls.

CONTROLLING M1

M1's base uses a two-wheel "wheelchair" drive. Connected to each wheel by a toothed belt and sprocket combination is a NEMA 23 frame stepper motor rated at 6.0 V and 1.0 A. There is also a third, unpowered caster wheel.

An SGS-Thomson L297/L298 stepper motor bipolar chopper drive powers the NEMA 23 motors with current limited to 300 mA. Steep accelerations and decelerations are possible even at this low current setting. Three NiCd batteries supply 30 V to the chopper drive, which gives the step rate an upper limit in excess of 6000 half-steps per second.

Stepper motors enable very accurate drive control, and this particular implementation appears to result in good performance and low power consumption at both low and high speeds. At 30 V, the batteries have a storage capacity of 600 mAh.

The "step now" input on each stepper motor driver is connected to a TPU line, so we can control the speed of each motor independently and precisely.

One problem with stepper motors is that they stall if you try to run them too fast or accelerate or decelerate too quickly. M1 has no stall-detection sensors, but it does have stall-recovery software. If the control software decides that no progress has been made for long enough, it will slow to a stop, which recovers from the stall.

Of course, it's much better to avoid stalls in the first place. M1 contains a software layer between the high-level control and the motors for this purpose. When the high-level control software commands a speed, this low-level software smoothly accelerates or decelerates to this speed, within the motors' safety parameters.

Internally, two sets of commands control wheel speed. One set controls the left and right wheel speeds independently. The other commands control the angular and forward velocities.

The mapping between the two command sets is simple. If *a* is angular velocity, *f* is forward velocity, and *l* and *r* are the left and right wheel velocities, respectively, then the mapping is:

$$l = jf - ka$$
$$r = jf + ka$$

where *j* and *k* are constants that depend on the units used.

GATHERING THE BALLS

M1's basic operation during the contest is to find a ball, grab it, carry it to the goal, and drop it in. And as we mentioned, there are two kinds of balls in the contest—standard tennis

Listing 1—*M1 iterates through all detected objects that are the color of the tennis balls or squiggle balls. After determining ball position, M1 can decide which to pursue.*

```
int find_targets(Target *dest, Vstate *vs, enum target_type type){
  Blob *blobs[MAX_TARGETS];
  int n_blobs;
  int i;
  int n_targets;
  /* Find largest MAX_TARGETS blobs on current channel */
  n_blobs= blobs_select_largest_n(blobs, frame_eb(vs), MAX_TARGETS);
  for (i= 0, n_targets= 0; i< n_blobs; i++){
    /* Find center of gravity of current blob */
    blob_find_cg(blobs[i]);
    /* If blob is too far left, too small, or over horizon, skip */
    if (blobs[i]->xcg < *p_track_min_col ||
        blobs[i]->area < ((*p_diam_thresh) * (*p_diam_thresh)) ||
        !camera_to_world(blobs[i]->xcg, blobs[i]->ycg,
        m1_camera_pos, &(dest[n_targets].x), &(dest[n_targets].y)))
      continue;
    dest[n_targets].area= blobs[i]->area;
    /* Set angle and distance, given robot-relative x-y coordinates */
    rect_to_polar(dest[n_targets].x, dest[n_targets].y,
                  &dest[n_targets].angle, &dest[n_targets].dist);
    /* Use perceived size and computed dist. to compute real size */
    dest[n_targets].size=int_sqrt(blobs[i]->area) *
         [dest[n_targets].dist / 1000;]
    /* If actual size is too small, ignore object */
    if (dest[n_targets].size < *p_size_thresh){
      continue;}
    dest[n_targets].type= type;
    dest[n_targets].score= 0;
    dest[n_targets].age= 0;
    n_targets++; }
  for (i= n_targets; i< MAX_TARGETS+1; i++){
    dest[i].size= 0;}
  return n_targets;}
```

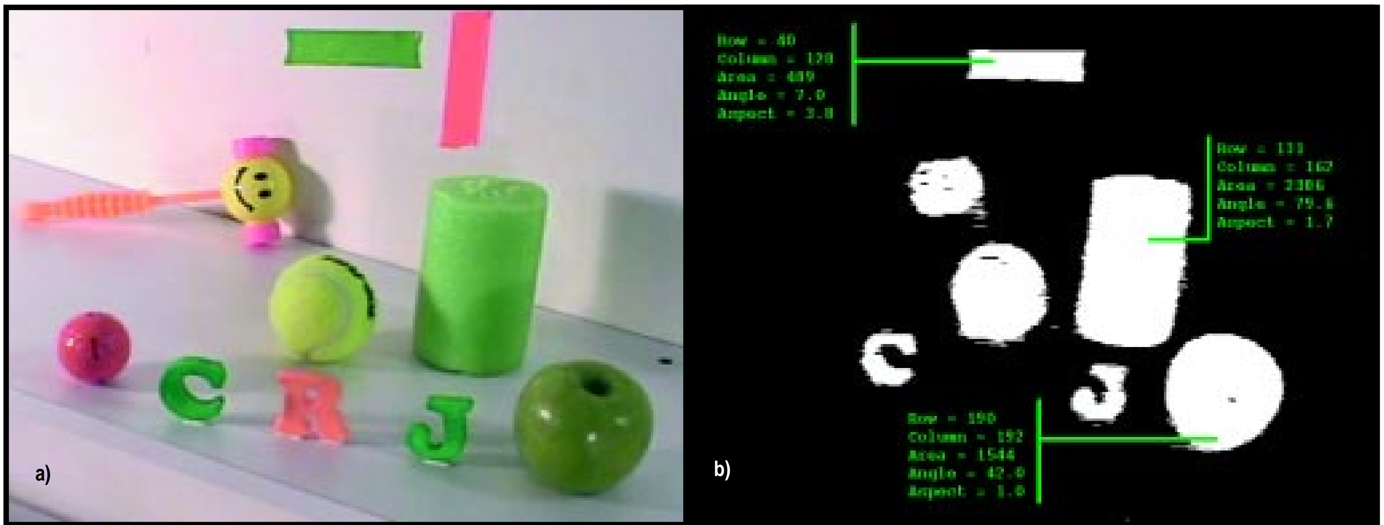


Photo 1—*There are two sides to every story, or in this case, two ways to view the same picture. These brightly colored objects (**a**) change appearance when seen by the Cognachrome vision system (**b**). The system assembles contiguous pixels of interest into blobs and calculates various statistics, such as centroid, area, elongation or aspect ratio, and direction of orientation. Notice that Cognachrome only sees colors it was trained on.*

balls and motorized, randomly-moving squiggle balls.

Of course, the real challenge is the squiggle balls. The squiggle balls are almost as big as M1's gripper and they move almost as quickly as M1, so the robot control must be very accurate to turn toward the squiggle ball and run it down. Once we can do that, it isn't hard to handle tennis balls as well.

The contest rules also require us to announce when M1 is chasing a squiggle ball. This is done via a small piezoelectric speaker that beeps when M1 sees a squiggle ball. Once the announcement is made, M1 chases the squiggle ball until it catches it or doesn't see it any more. Tennis balls are ignored to make it clear to the judge that M1 really is distinguishing tennis balls and squiggle balls.

LOCATING THE BALL

The tennis balls are greenish yellow, and the squiggle balls we use are red. We train two of the Cognachrome's three color channels on these colors.

When the Cognachrome detects the ball color, it reports the x and y coordinates of the ball's center, relative to the camera's field of view. These numbers need to be translated into a rotation angle and distance.

The control software uses the angle to decide how to turn and uses the distance to determine the ball's location relative to the gripper. The function definition is shown in Listing 1.

The translation involves some interesting math. It is handled in two stages, which we will present in reverse order.

PERSPECTIVE TRANSLATION

First, imagine that the camera gives us a nice perspective image, something like Figure 2a. From an image like this, we can compute the distance and angle to the ball straightforwardly (assuming that the ball is on the ground).

In Figure 3a, ϕ is a straightforward function of the y coordinate of the ball and the tilt of the camera. (M1 can tilt the camera with a stepper motor, so the ball-location routine has to compensate for this.)

We know h—it's the location of the camera above the ground (~8″ for M1) minus the height of the center of the ball. To find x, we use:

$$x = h \tan(\phi)$$

We also want to know d:

$$d = \sqrt{x^2 + h^2}$$

Figure 3b looks like a top view, but it's actually looking from a little bit forward of top (compare it to Figure 3c). We are looking from the direction labeled with the arrow in Figure 3a.

Here, θ is a straightforward function of the x coordinate of the ball, and d was computed above. To find y, use:

$$y = d \tan(\phi)$$

Now we have x, which is the distance from the camera forward to the ball, and y, the distance left or right to the ball. What we want is the angle to turn and to head toward the ball (when the turning point is centered between the two drive wheels) and the distance to the ball.

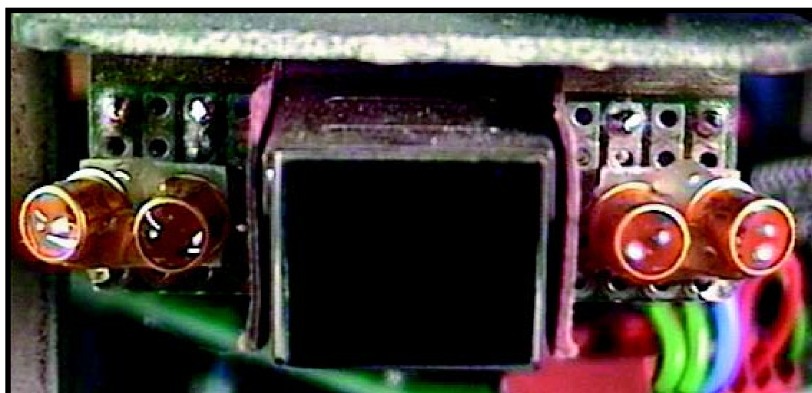


Photo 2—*The left half of M1's infrared sensor array is composed of a Sharp GP1U52X infrared detector sandwiched between four infrared LEDs.*

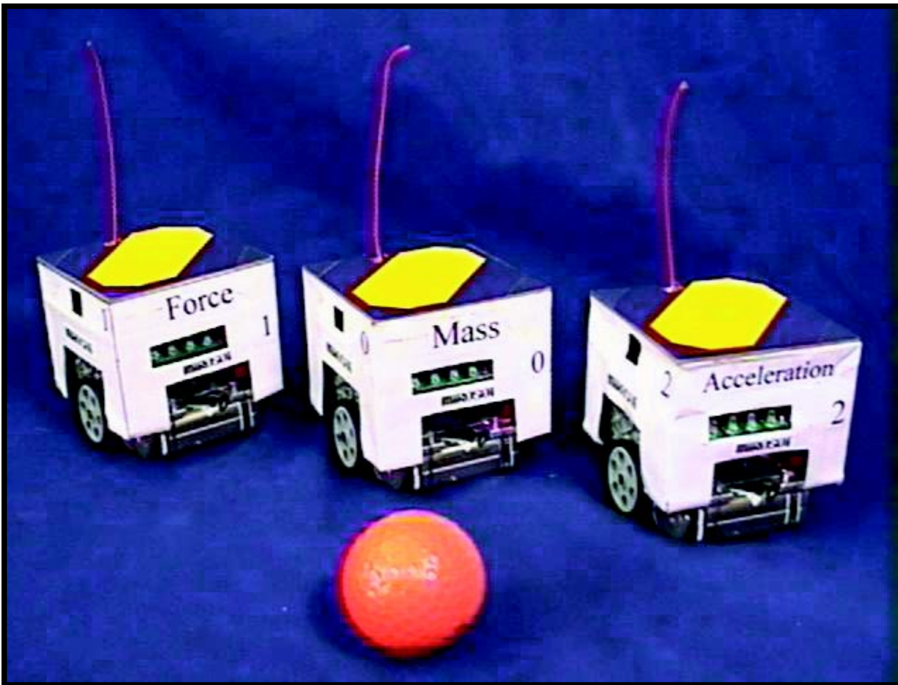


Photo 3—*Force, Mass, and Acceleration are the three members on Newton Labs' world-champion robot soccer team. (Mass is the goalie.) In the foreground is the soccer ball (actually an orange golf ball.)*

So finally, the distance to the ball is:

$$\sqrt{(x+c)^2 + y^2}$$

and the angle is:

$$\psi = \tan^{-1}\left(\frac{y}{x+c}\right)$$

FISH EYES

Unfortunately, this isn't the whole story. Remember our assumption that the camera gives a nice perspective image? It doesn't.

To get the right compromise between peripheral sensing and seeing in the distance, we use a camera with about a 100° field of view. This results is a serious fish-eye effect—the nice, straight lines in Figure 2a look curved when viewed through the camera (Figure 2b).

We need to find a mapping that undoes this fish-eye distortion before applying the above mathematics. Basically, this mapping should use the *x* and *y* coordinates from the vision data to compute the θ and ϕ angles suitable for use in the equations.

When we implemented this code, we tried to derive the correct mathematical form of the mapping. We soon decided it would be easier to approximate it. We used polynomial equations because they're easy to deal with.

A linear mapping like $\theta = axy + bx + cy + d$ is not sufficient. We need a slightly more complicated polynomial—a bivariate quadratic. We suspected this type would be adequate because the curved lines produced by the fish-eye effect look vaguely like parabolas.

However, if it had not been adequate, we had to be prepared to move on to higher-degree polynomials or find a different form of equation. Therefore, we wanted to find values for the coefficients *a*–*r* in:

$$\theta = ax^2y^2 + bx^2y + cx^2 + dxy^2 + exy + fx + gy^2 + hy + i$$

$$\phi = jx^2y^2 + kx^2y + lx^2 + mxy + nxy^2 + ox + py^2 + qy + r$$

First, we needed some experimental data. We set up a vision target as far as possible from M1 and had the robot pivot from side to side and rotate the camera up and down in a predefined grid pattern.

At each location, we recorded the *x* and *y* positions of the target according to the vision system as well as the vertical and horizontal angles, based on how far the robot pivoted and rotated its camera.

We then needed to find the values for *a*–*r* that would minimize the error between the computed θ and ϕ values and the measured values. Although this task may sound daunting, we simply plugged all the values into an Excel spreadsheet, calculated the differences for each sample, and summed the squares of the differences.

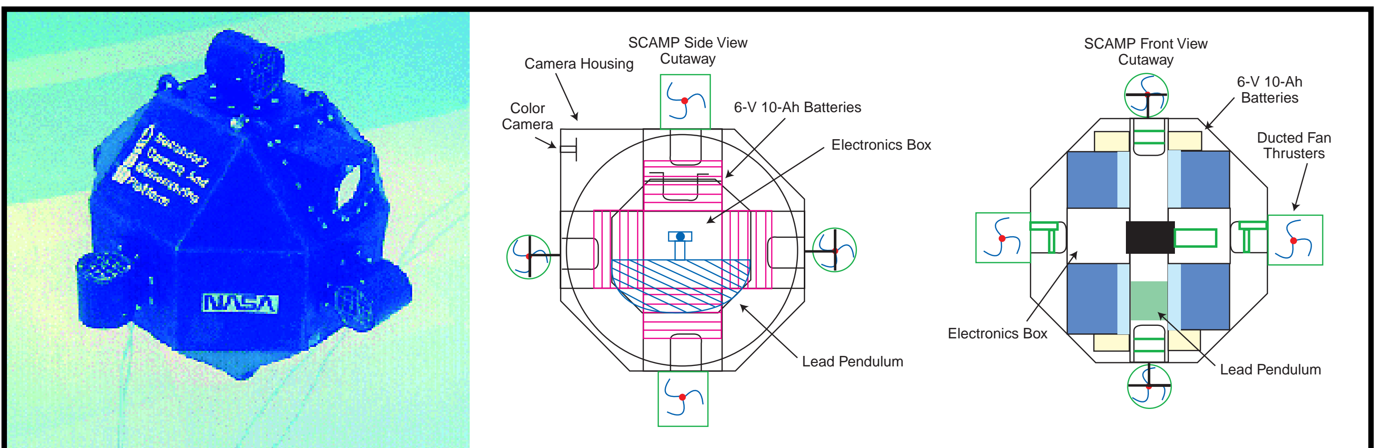
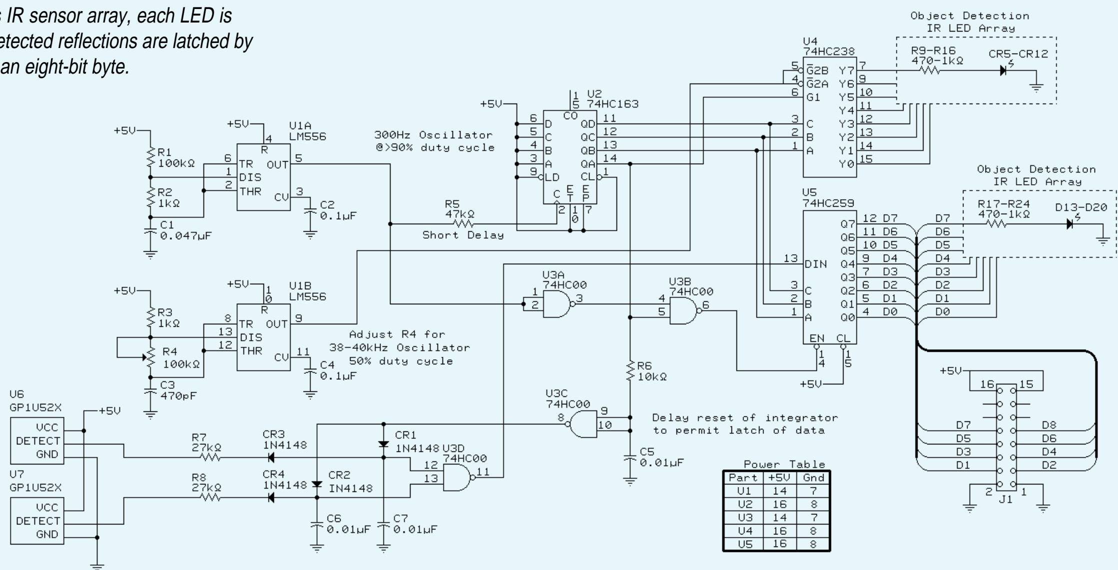


Photo 4—*The SCAMP underwater robot, created by the University of Maryland's Space Systems Laboratory, is designed to simulate zero-gravity spacecraft motion. With the use of a Cognachrome vision system, SCAMP can autonomously perform simulated docking and station-keeping maneuvers.*



Part	+5V	Gnd
U1	14	7
U2	16	8
U3	14	7
U4	16	8
U5	16	8

Figure 4—*In M1's IR sensor array, each LED is fired in turn and detected reflections are latched by the 74HC259 into an eight-bit byte.*

We let Excel's Solver find values for *a*–*r* that minimized this error sum. (The Solver isn't installed by default, so you might need to find your installation CD to add this feature.)

GRABBING THE BALL

Thanks to all the above math, M1 now knows the distances and angles to all the balls in view. The next task is to choose a ball and chase it down, where the chase is a lot easier for a tennis ball than a squiggle ball.

We already mentioned that once M1 starts to track a squiggle ball, it continues tracking it until the ball is within reach or disappears from view. Also, once M1 starts tracking a tennis ball, it does not switch to a squiggle ball unless the squiggle ball is about half as far away as the tennis ball.

We use the following algorithm to head for a ball, given an angular offset, ψ. Here, a is the required angular velocity, e is an angular error term, and f is the required forward velocity:

$$a = k_1\psi$$
$$e = k_2\psi^2$$
$$f = sk_3(1 - e)$$

(If $e > 1$, then we set the speed to zero, rather than moving backward.)

The constant s has different values for tennis balls and squiggle balls. M1 moves as quickly as possible to chase squiggle balls. But, it's more cautious when approaching tennis balls because they have a tendency to bounce off the gripper and roll away.

We quickly found that this algorithm doesn't work all the time. If a ball is within reach but to the left or right of the gripper, M1 pivots toward the ball and the gripper then knocks the ball away. So, we use a different algorithm for this situation—M1 simply backs up.

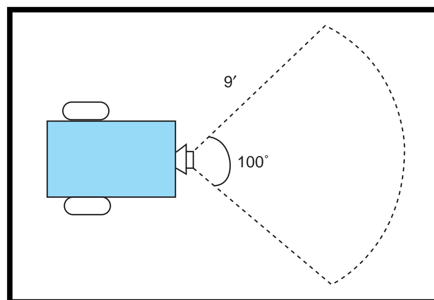


Figure 5—*M1's camera can detect balls in a pie-shaped region.*

SEARCHING FOR BALLS

If M1 cannot see any balls at the moment, it has to find some. When M1 starts looking for balls, it first spins around to try to see one. However, that doesn't always work. The repository might be in the way. Or, if the balls are too far away, M1 can't see them.

If a simple spin doesn't find any balls, M1 goes searching. It heads forward until it finds a wall (unless it finds a ball), and then it follows the wall.

M1 follows the wall using an infrared obstacle detector. The code drives two banks of four infrared LEDs one at a time, each modulated at 40 kHz.

Two standard Sharp GP1U52X infrared remote-control reception modules detect reflections. The 74HC163/74HC238 combination fires each LED in turn, and the 'HC259 latches detected reflections. This system provides reliable obstacle detection in the 8–12″ range. Figure 4 shows the schematic, and Photo 2 shows the IR sensors.

The system provides only yes/no information about obstacles in the eight directions around the front half of the robot. However, M1 can crudely estimate distance to large obstacles (e.g., walls) via patterns in the reflections. The more adjacent directions with detected reflections, the closer the obstacle probably is.

SEARCHING THE ENTIRE REGION

Unfortunately, even with M1's wide 100° field-of-view camera (illustrated in Figure 5), wall following doesn't cover the whole room. It just sees the areas depicted in Figure 6a.

So, every few seconds, M1 stops, spins 180° away from the wall, then spins back to the original direction. This sequence enables it to see into the center of the room from various points along the wall (see Figure 6b). M1 does this once every 8 s in the first 8 min. of the round, and once every 4 s in the final 2 min.

DUMPING THE BALL

Once M1 has the ball, it must dump it in the repository. Contestants can build their own ball repository, and we marked ours with a blue rectangle.

To keep the squiggle balls inside, we put a 1″ lip in the repository's gate, so the gripper has to lift the balls over this lip to deposit them. However, M1 would go after the balls in the repository if it could see inside, so we covered the gate with a black curtain and put the blue marker on the curtain.

Much like searching for a ball, M1 starts its search for the repository by spinning. If it doesn't see the blue marker, it heads for a wall and follows it around.

When it sees the blue marker, M1 heads straight for the repository. It begins to slow down and slows down even more as it nears the repository.

The size of the blue marking is used to estimate the distance. We can't use the vertical angle to the marker, like we do for the balls, because the rectangle is at roughly the same height as the camera.

Two bump sensors on the bottom of the gripper tell M1 when it reaches the lip of the gate. They also enable M1 to line up with the gate before it drops the ball.

When one bump sensor is engaged, M1 turns off the wheel on that side and turns on the wheel on the other side. This action causes M1 to line up with the gate. M1 drops the ball when both bump sensors are engaged.

Fashion collided with function when one of the spectators wore a bright blue shirt in a preliminary round. The shirt was approximately the same color as the gate marker, and the spectator stood next to the 3′ wall surrounding the playing field. When M1 picked up a ball, it often headed straight for the spectator rather than the repository. Not able to reach the repository, the robot acted quite confused.

We fixed this problem by computing the vertical angle to the gate marker (using the same algorithm, including

fish-eye correction, as for the balls) and ignoring blobs above a certain angle.

We had already compensated for a similar problem by ignoring red and yellow blobs above the horizon. Otherwise, M1 might have viewed certain spectators as huge squiggle balls.

M1's control software is surprisingly complex given its seemingly simple task. While describing the entire software system is outside the scope of this article, the state diagram in Figure 7 gives you the overall picture.

GAME DAY

We worked through the night before the contest, tweaking the algorithms. Early the morning of the contest, M1 completed three perfect runs. We called it complete then and froze the code.

To add a little extra stress, the competition was being recorded for Scientific American Frontiers with its host, Alan Alda, standing next to the arena giving commentary. M1 got off to a strong start, capturing the first tennis ball in mere seconds. It continued roaming around the arena and quickly collected almost all the tennis balls and both squiggle balls.

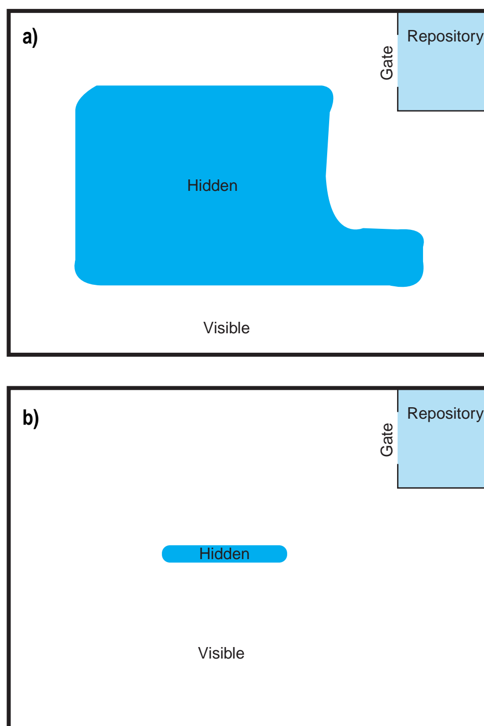


Figure 6a—*If M1 searches the arena simply by following the wall, it misses most of the middle.* ***b—****If M1 periodically pivots to face the middle of the room while following the wall, it searches a much larger region.*

However, the final tennis ball remained elusive. It was in the exact center of the arena, and remained just slightly beyond the visual reach of M1 as it scanned the arena. Clearly, to collect this ball, M1 had to turn and look into the center of the arena from exactly the right point along the wall.

The spectators grew tense as M1 followed the wall around and around, turning and looking toward the ball but not quite seeing it. Time was running out.

Finally, on its third time around the arena, M1 looked into the center from just the right spot, collected the ball, and sped to the repository with seconds to spare, earning a perfect score. The crowd erupted into cheers and applause. And, the Newton Labs team began to breathe again.

ROBOTS SEE THE WORLD—AND BEYOND

The Cognachrome vision system serves a wide range of applications, from research uses like catching balls, autonomous spacecraft docking, and automated acquisition of cargo by helicopter, to industrial uses like sorting fruit and inspecting upholstery.

We entered (and won) the first and second International Micro Robot World Cup Soccer Tournaments (MIROSOT) held by KAIST in Taejon, Korea, in November of 1996 and June of 1997. We used the Cognachrome system to track our three robots' position and orientation, the soccer ball, and the three opposing robots. Our team is pictured in Photo 3.

Because of the robots' small size (each fits into a 7.5-cm cube), we opted for a single vision system connected to a camera over the field instead of a system in each robot. (In fact, the rules of the contest require markings on the top of the robot that encourage this. All but one of the teams used a single camera above the playing field.)

Professor Jean-Jacques Slotine and Dr. Kenneth Salisbury of MIT incorporated two Cognachrome systems into their adaptive robot arm—the WAM (whole arm manipulator). Using two-dimensional

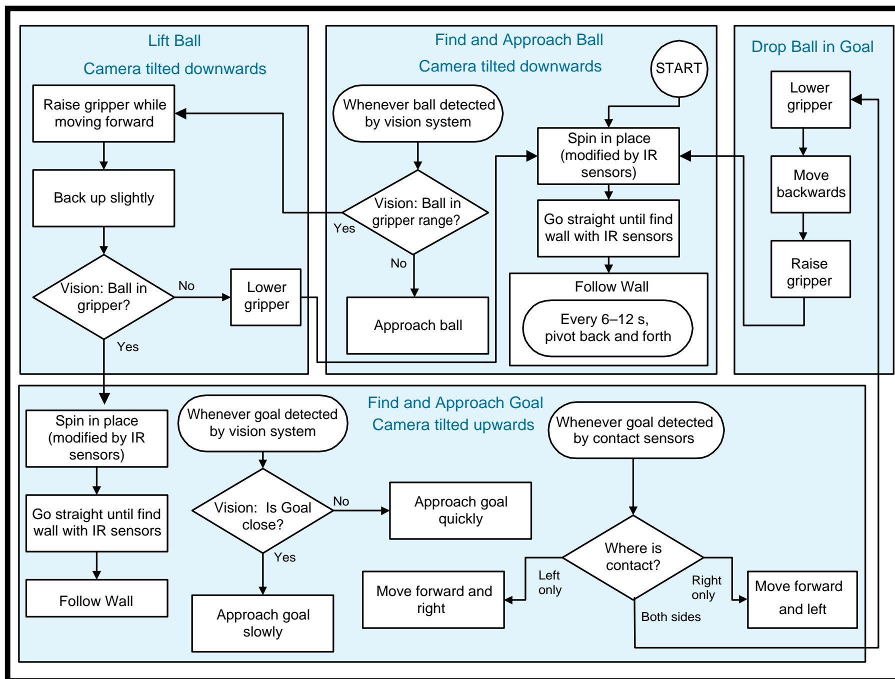


Figure 7—*This diagram gives a simplified view of M1's different behavior states and how they are activated. Not shown here are special time-out behaviors designed to get M1 unstuck if it hasn't made progress for some time.*

stereo data from a pair of Cognachrome systems, the WAM controller predicts the three-dimensional trajectory of a ball in flight and controls the arm to quickly intercept and catch the ball.

The University of Maryland Space Systems Laboratory and the Kiss Institute for Practical Robotics have simulated autonomous spacecraft docking in a neutral buoyancy tank for inclusion on UMD's Ranger space vehicle. Using a composite target of three brightly-colored objects designed by Dr. David Miller, the spacecraft (shown in Photo 4) knows its distance and orientation and can servo to arbitrary positions around the target.

So, although participating in the AAAI contest was exciting, what it really demonstrated is that robots can perform interesting tasks using a simple, fast vision system. ■

Bill Bailey is a design engineer at Newton Research Labs, a company that develops high-performance, low-cost machine vision hardware and software for industrial and robotic applications. The original developer of the M1 robot base, Bill has over 25 years of expertise covering analog and digital electronics, software, and mechanical design. He and the other authors may be reached via vision@newtonlabs.com.

Jon Damon Reese received a B.A. in computer science from Rice University and an M.S. and Ph.D. in information and computer science from the University of California, Irvine. His research interests over the years have included artificial intelligence, programming languages, software engineering, and software safety. Jon serves as a software and applications specialist at Newton Research Labs.

Randy Sargent is the president of Newton Research Labs. He received a B.S. in computer science at MIT, and an M.S. in media arts and sciences from the MIT Media Laboratory. Formerly holding titles of Lecturer and Research Specialist at MIT, he is one of the founders of the MIT LEGO Robot Contest (a.k.a. 6.270), now in its ninth year.

Carl Witty is a research scientist at Newton Research Labs. He received his B.S. and M.S. in computer science from Stanford University and MIT, respectively. A member of the winning team in the 1991 international ACM Programming Contest, his interests include robots, science fiction and fantasy, mathematics, and formal methods for software engineering.

Anne Wright is the senior design engineer at Newton Research Labs. The primary architect of the Cognachrome vision system, Anne received her B.S. and M.Eng. in computer science from MIT. She also helped lead and develop technology for the MIT LEGO Robot Contest from 1992 to 1994.

SOFTWARE

Design documentation for M1, including the full source code, is available at www.newtonlabs.com/cc.html#ml.

A video tape highlighting applications discussed in the paper (e.g., M1 picking up tennis balls, the soccer robots performing, etc.) can be ordered from www.newtonlabs.com/cc.html#video.

REFERENCES

www.newtonlabs.com/cc.html
www.mirosot.org
www.ai.mit.edu/projects/wam/index.html#S2.2
www.pbs.org/saf/8_resources/83_transcript_705.html

SOURCES

Cognachrome vision system
Newton Research Labs
Robotics Systems and Software
14813 NE 13th St.
Bellevue, WA 98007
(425) 643-6218
Fax: (425) 643-6447
www.newtonlabs.com

GP1U52X
Sharp Electronics Corp.
Microelectronics Group
5700 NW Pacific Rim Blvd., Ste. 20
Camas, WA 98607
(360) 834-2500
Fax: (360) 834-8903

L297/L298
SGS-Thomson
55 Old Bedford Rd.
Lincoln, MA 01773
(617) 259-0300
Fax: (617) 259-4421

I R S

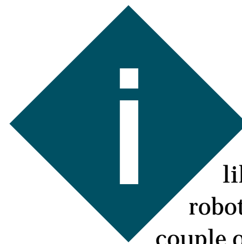
401 Very Useful
402 Moderately Useful
403 Not Useful

FEATURE ARTICLE

Ingo Cyliax

Power Systems in Autonomous Robots

Power presents a special challenge when it comes to robot design. After reviewing the various types of batteries, Ingo shows how he untethered the power system on the Stiquito robots he introduced in an article in *INK* 73.



It's pretty clear I like to dabble in robotics. Over the past couple of years, I've written about controllers for six-legged walking robots (*INK* 73) as well as robot navigation schemes (*INK* 81).

This time, I want to zero in on some of the power systems used in robotics and take a look at how the power system for the Stiquito II was upgraded to an untethered system. Previously, we used a novel bumper-car–type power-delivery system to run these micro-robots. (You can take a look at a Stiquito II on the cover of *INK* 81.)

But before looking at the Stiquito power system, I want to tell you about some of the design issues involved with robot systems.

After giving a short overview of potential power sources, I demonstrate how you can adapt power supplies to the power requirements of a typical robot's subsystems.

DESIGN TRADEOFFS

When you're designing a power system for mobile robots, the biggest concern is power density. In a nutshell, power density is a figure of merit describing the amount power you can expect for a given weight or volume.

Power density is usually represented by expressing the power system's capacity in watt-hours compared to some measure of weight or volume. Common units are Wh/lbs., Wh/kg, and Wh/l.

The capacity of a power source is typically represented by the amount of current or power the system can provide over time. Units such as ampere-hours or watt-hours are used here.

For example, a battery may be rated at 12 V and 1200 mAh. That is, it can provide 1.2 A at 12 V for 1 h. Of course, if the current is drawn at 120 mA, it should last 10 h.

Battery capacity usually varies by the discharge rate as well. So, the figure of 1200 mAh may only be good when discharged at 120 mA, and it may be lower if discharged at a higher rate. That is, it may only have 900-mAh capacity when discharged at 1.2 A.

Another design decision is whether to use rechargeable (e.g., batteries) or replaceable (e.g., dry-cells and fuels like gasoline or hydrogen) power sources.

Finally, whatever power source you use, you have to worry about conversion. As you know, whenever energy is converted, a little bit is lost. So, matching the power source to the type of loads in the system is also a design tradeoff.

For example, if you want to build an autonomous flying robot, it may not make sense to use electrical energy as a primary energy for the craft and use electric motors for propulsion. It may be much more efficient to power the craft with a combustion engine to provide propulsion and a small generator to power electrical components like the computer.

POWER SOURCES

There are a variety of power sources to choose from. Power sources are classified into categories by how the energy is produced.

Well, technically, energy is never really produced; it's converted. So in all these cases, power sources are just devices and systems that convert energy stored in one form into energy we need.

For robotics, the most convenient form of energy is electrical energy. Electrical energy is easy to manage,

and we need it to power our logic anyway. So, let's look at some of the power-storage and -conversion systems we can use for robots.

The most common power source is the electrochemical battery. The term "battery" refers to a collection of cells, which is the basic unit that power is converted in. Cells are connected in parallel to increase current and in series to increase output voltage.

David Prutchi gives an excellent overview of various battery chemistries in "Battery-operated Power Supplies" (*INK* 55). Table 1 summarizes some of the most common primary and secondary battery technologies and their energy density and cell voltages.

Fuel cells are another type of electrochemical cell. In a fuel cell, the agents are fed into the cell, and the reaction is maintained as long as fuel and oxidizer are provided.

Fuel cells have potentially high power density, since the non-energy-producing elements of the cell, the electrolyte, and mechanical construction are continually reused. Also, they are reliable since they contain no moving parts. They also provide no nasty combustion by-products. For example, the hydrogen-oxygen fuel cell produces water.

Today's fuel cells typically operate using hydrogen as the fuel and oxygen as the oxidizer. Small fuel cells can power small electric cars and buses, as well as provide electrical power for spacecraft. Some day, fuel cells may operate from other fuels besides hydrogen, which will make them versatile.

Another popular power source is the solar cell. A solar cell is a photovoltaic system that uses PN-junction semiconductors (diodes). All semiconductor diodes are inherently photosensitive. However, the trick is making these diodes sensitive to sunlight, rather than infrared light, and large in area so they produce enough current.

Solar cells aren't very efficient. They achieve ~20% efficiency in converting the 100–250 W/m^2 of sun energy that can reach Earth's surface on a sunny day.

The solar cell's open-circuit voltage is 0.5 V. Several cells are typically wired in series to give more useful outputs levels. Solar cells provide constant current for a particular light level, which makes them suitable for charging secondary batteries.

Battery Type	Anode	Cathode	Vwork	Wh/kg	Wh/l
Primary Type					
Leclance	Zn	MnO_2	1.2	80	140
Magnesium	Mg	MnO_2	1.5	125	195
Alkaline	Zn	MnO_2	1.3	95	210
Mercury	Zn	HgO	1.2	95	325
Mercad	Cd	HgO	0.85	45	180
Silver oxide	Zn	Ag_2O	1.5	130	515
Zinc-air	Zn	O_2	1.2	290	905
Li-SO_2	Li	SO_2	2.9	340	440
Li-MnO_2	Li	MnO_2	3.5	200	400
Secondary Type					
Lead-acid	Pb	PbO	2.0	40	80
Edison	Fe	Ni NiO	1.2	40	60
Nickel-cadmium	Cd	Ni NiO	1.2	50	80
Silver-zinc	Zn	AgO	1.5	140	180
Nickel-zinc	Zn	Ni NiO	1.6	70	120
Nickel-hydrogen	H_2	Ni NiO	1.2	55	60
Silver-cadmium	Cd	AgO	1.1	60	120
Zinc-chlorine	Zn	Cl_2	1.9	100	130
Nickel-metal hydride	H2 (metals)	NiOOH	1.2	60	
Lithium/Aluminum iron disulfide (400°C)	Li/Al	FeS_2	1.4	100	100
Lithium/Aluminum iron sulfide (400°C)	Li/Al	FeS	1.2	80	100
Sodium sulfur (300°C)	Na		1.9	100	150
Zebra (200°C)	Na	$NiCl_2$	2.4	100	150

Table 1—*Let's compare the working cell voltage and capacity density of the most popular battery chemistries. As you can see, some batteries only operate at high temperatures.*

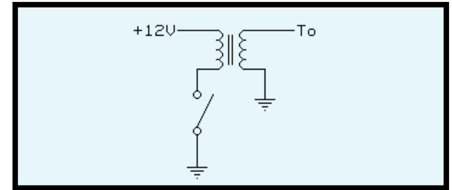


Figure 1—*In this simplified ignition coil, when the switch is closed, energy is stored as a magnetic field. When you open the switch, a high voltage is generated in the secondary as the field collapses.*

Electromechanical systems convert mechanical energy into electrical energy. Some typical electromechanical devices are generators and alternators.

Electric motors convert electrical energy into mechanical energy. One typical application of electromechanical system is to use an internal combustion engine to power a generator for electrical power.

When self-contained, these systems are called gen-sets. The energy is stored in the fuel for the engine. Robotic vehicles besides cars have used this method to generate electrical power for robots. Also, vehicles that use internal combustion engines for propulsion usually have a generator to produce electrical power.

Another novel electromechanical system which promises to have good efficiency for storing energy is the flywheel. A flywheel is a mass which stores energy in the form of inertia by spinning the mass at a high velocity.

A dual-function electric motor-generator is incorporated into the flywheel unit. The motor is used to accelerate the flywheel (i.e., to store energy). Electrical energy is provided by consuming the inertia to power the generator. Discharging the energy stored in a flywheel slows the rotation of the mass.

Lightweight flywheel systems are being developed for mobile applications. These systems use carbon fiber composite for the flywheel mass.

The flywheels are encased in a vacuum enclosure to eliminate friction from air, and they are levitated with magnetic bearings, which also act as the motor-generator units. To achieve the high energy density desired in vehicles, the flywheel is then spun to very high speeds. These systems are being developed for electric cars and buses.

Thermoelectric devices are usually solid-state junction devices which convert temperature differentials into electric energy. They're not very efficient but can be used when abundant heat is available.

One application is interplanetary space probes. These probes use radioactive power generators (RPGs) to convert the heat generated by a thermonuclear reaction to generate electrical energy. Of course, because they're in space, they don't need to be shielded well, which makes them light. Also, since no moving parts are involved, these systems are reliable—a must when you can't service your robot.

CONVERSION TECHNIQUES

Now that you're up to speed on the various electrical power sources, here comes the hard part.

Power requirements in a robot are typically diverse. At a minimum, power is required for actuators or propulsion, the logic controlling the robot, and possibly a communication system.

These subsystems have different voltage and current requirements. For example, the logic might require a regulated 5-V power supply, while a radio module might need unregulated 12 V. Of course, the propulsion system has different requirements, typically at high currents.

Let's look at some conversion techniques for electrical energy that enable us to adapt our power source to different subsystems. For us, DC-to-DC converters are the most common. These converters transform the input voltage to output voltage.

There are three kinds of DC-to-DC converters—step down, step up, and voltage inverter. The step-down converter is probably the most common. It converts a high-voltage power supply to a low-voltage one.

Similar to the step-down, the step-up converter steps up the input voltage to the output voltage. The last type, the inverter, converts positive to negative or negative to positive voltage.

While all this is pretty convenient, remember when we convert voltages, we don't create power. This might seem obvious, but it's easily forgotten.

The output power of any converter is going to be less than the input power. That is, we lose power, usually in the form of heat in the converter. The efficiency of a converter is expressed as a percentage:

$$E = \left(\frac{P_{out}}{P_{in}}\right)100\%$$

The power-converter efficiency lets us calculate the total power requirements of the system by inflating the power used by power sources when adapted by a converter.

For a low-power application, like you usually encounter in a robotics system (unless it's a car-sized mobile robot), two kinds of converter techniques are commonly used—switched inductor and switched capacitor (i.e., flying capacitor) converters.

The switched inductor converter is the most versatile and efficient, so let's look at it first.

SWITCHED INDUCTOR CONVERTER

The basic idea of a switched inductor converter is to use the inductor's properties of storing energy in the form of a magnetic field to adapt input and output impedances. An inductor stores energy by building up a magnetic field, which is generated by current passing through it. When the current flow is interrupted, an inductor uses the stored magnetic field to try to maintain current flow.

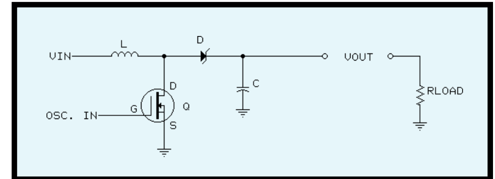


Figure 2—*In this step-up converter, the MOSFET transistor (Q) sets up a charging current through the inductor (L). When a transistor isolates the inductor, the inductor discharges its stored energy through the keeper diode (D) into the load (R_{load}).*

I like to think of an inductor as something like a current capacitor. As the inductor tries to maintain the current, it induces a voltage across it. The voltage induced is the voltage needed to push the current it's maintaining through a load. The current, of course, decays as the energy stored in the field is used up to push the current. The voltage is thus related to the rate of discharge by:

$$V = L\left(\frac{dI}{dt}\right)$$

For example, if you take an inductor and connect it to a power supply, the current builds up over time as it builds the magnetic field until a steady-state current is reached. If you now disconnect the inductor, it tries to maintain the current by inducing a voltage across it. Since the inductor is not connected to anything, it induces a high voltage to push the current through a high impedance (i.e., air).

In principle, this is how a car ignition works. The inductor builds up a very high voltage—enough to break down the fuel-air mixture in the cylinder between the spark gap of the spark plug.

The car ignition also uses a step-up winding, which is magnetically coupled to the primary winding to further increase the output voltage generated when the current is cut to the primary winding (see Figure 1).

How does this apply to converter circuits? If you think about it, a car ignition system is just a step-up converter. And in fact, that's how high-voltage step-up (also sometimes called fly-

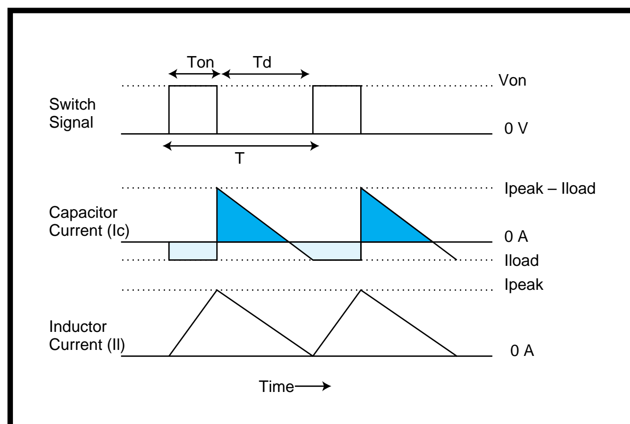


Figure 3—*This timing diagram shows what's going on in the circuit in Figure 2. You can see the transistor switching signal and the current through the capacitor (Ic) and inductor (Il).*

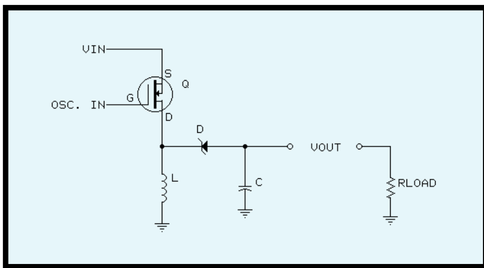


Figure 4—*This configuration inverts the input voltage—something that can't be done with a linear power supply.*

back) converters work. Let's now look at how this can be applied to DC-to-DC step-up converters.

By using an inductor, we can generate higher induced voltages than the voltage used to establish the current through the inductor. However, as the inductor discharges, the energy stored in the magnetic field is used up, eventually dissipating and needing to be reinstated.

This means we have to periodically connect the inductor to our input power supply to set up the field and then connect it to the load to dissipate the energy. We do this with a circuit like Figure 2.

In Figure 2, a MOSFET (*Q*) switches the inductor (*L*) to ground, setting up a current through the inductor to establish the field. When the ground path is broken, the inductor induces an output voltage. When the induced voltage is higher than that stored in the capacitor, the catcher diode (*D*) lets the current discharge into the filter capacitor and the load (R_{load}).

Once the inductor has spent its energy, the cycle is repeated by connecting the inductor to ground. The capacitor is important in this circuit, since it needs to provide current to the load when the inductor is being charged, because the voltage across the inductor then is only as high as the input supply. Figure 3 shows the signals in this circuit.

The switching signal is the key to this converter. It needs to meter how much energy is stored in the inductor delivered to the load. If too much is delivered, the voltage starts to rise and may become very high.

Fortunately, it's easy to calculate the relationship of this signal. The duty cycle of on- and off-times is simply the ratio of the boost voltage ($V_o - V_{in}$) to the output voltage (V_o):

$$DC = \frac{T_{on}}{T_d} = \frac{V_o - V_{in}}{V_o}$$

This is easy to remember, since the duty cycle is zero when the input voltage is as high as the desired output voltage. The size of the inductor depends on the desired output current, input voltage, and on-time (T_{on}) of the MOSFET.

$$L = \frac{V_{in} \times T_{on}}{I_{peak}}$$

where the maximum current in the inductor (I_{peak}) is:

$$I_{peak} = \frac{2 \times I_{load} \times V_o}{V_{in}}$$

The capacitor (*C*) needs to be large enough to maintain the current to the load with an acceptable amount of ripple (V_{ripple}) voltage:

$$C = \frac{\Delta Q}{V_{ripple}}$$

$$C = \left(\frac{\left(I_{peak} - I_{load}\right)^2}{2 \times V_{ripple} \times I_{peak}} \right) T_{on} \left(\frac{V_o}{V_o - V_{in}} \right)$$

So this pretty much lets us define the parameters for the components in a switching power supply. I'll show an example of this when I talk about implementing an untethered power supply for the Stiquito.

Similarly, we can design a step-down converter and a voltage inverter using switched inductance technique. The values for the components are arrived at similarly. Figure 4 depicts a voltage inverter configuration.

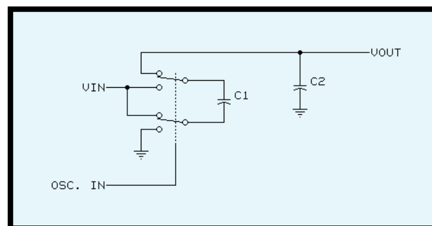


Figure 5—*Here, an analog DPDT CMOS switch connects the charge transfer capacitor (C1) between the input to charge and between the input and output to double the voltage.*

SWITCHED CAPACITOR

By contrast, in the switch-capacitor converter, you use a capacitor to transfer charges. The switched-capacitor converter is much simpler to think about and implement.

The essential mechanism in this type of converter is that we use a capacitor which is charged from the input power supply and then disconnected from the input power and connected in different configurations to arrive at the needed output voltage. Figures 5 and 6 show a step-up and an inverter configuration of a switched-capacitor converter.

It's easy to see you can only implement output voltages that are multiples of the input voltage. That is, the smallest voltage you can switch is the power-supply voltage. This, combined with the limited current capabilities due to internal resistance of large capacitors, limits the applications of the switched-capacitor converter.

However, they are popular for generating the bipolar power supplies (±10 V) needed for RS-232 implementation from a single 5-V supply. Also, they can be used for low-current high-voltage power supplies by cascading stages.

NiCd POWER IN STIQUITOS

In *INK* 73, I talked about controlling a small Nitinol-based robot, Stiquito II. The Stiquito uses Nitinol wires for actuators. The wires, sometimes called "muscle wires," contract when they're heated above a threshold temperature.

These wires aren't energy efficient (i.e., not much of the energy that is needed to heat the wire through I^2R heating is converted into mechanical energy). However, since it's easy to use them to build small and light actuators, they're common in miniature robotics and animatronics.

Since Nitinol wires use so much power, we originally used a sort of tethered system to power the robots. That is, we didn't carry any power on the robot itself.

This method was effective for powering the robots, but it made the system unwieldy. The tether system consists of a large cage, and the robots receive power from brushes, which connect them to an overhead screen and a copper floor.

To make Stiquitos more mobile, a small NiCd cell power supply powers the propulsion and logic systems on these robots. Even though NiCd batteries don't have the highest power density of all the secondary battery chemistries, they come in a variety of common cell sizes, all the way from button cells to D-size cells. Also, in-between sizes are available.

One popular cell size is the sub-C cell. This cell has the same length as a regular C-size battery, but it's skinnier. Sub-C NiCd cells have capacities of 1200 mAh. It turns out that two of these cells are the maximum that the Stiquito can carry.

Another reason NiCds are well suited for this job is the very low internal resistance of the battery. NiCds can generate very large currents, which is just the thing we need to power our Nitinol actuators.

Two cells generate a high enough voltage (2.4 V) to power the Nitinol actuators in the Stiquito directly using power MOSFETs to switch them, so they match the Nitinol load well. However, two cells aren't enough to power the logic on the Stiquito, which runs at 5 V and consumes up to 100 mA.

To generate the necessary 5 V, I use a step-up converter. The current requirement for the logic is too high to consider a switched-capacitor converter, so I use a switched-inductance converter.

To keep things small, I constrained myself to using a 22-mH inductor. So let's calculate the timing parameters for the switching signal to drive this converter. We can calculate the duty cycle (*DC*) by:

$$
\begin{aligned}
DC &= \frac{T_{on}}{T_{on} + T_d} \\
&= \frac{V_o - V_{in}}{V_o} \\
&= \frac{5 - 2.4\ V}{5} \\
&= 52\%
\end{aligned}
$$

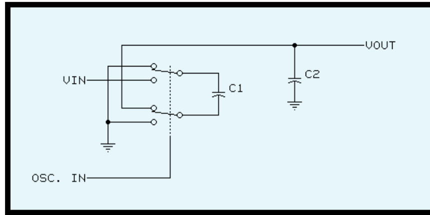


Figure 6—*Similar to the switched capacitor step-up converter in Figure 5, a switched capacitor converter can be used to invert voltages as well.*

Since the value of the inductor and currents are known, we can calculate T_{on} and thus the frequency of the switching signal:

$$
\begin{aligned}
I_{peak} &= 2 \times I_{load}\left(\frac{V_o}{V_{in}}\right) \\
&= \frac{2 \times 100\ mA \times 5\ V}{2.4\ V} \\
&= 0.417\ A
\end{aligned}
$$

$$
\begin{aligned}
T_{on} &= L\left(\frac{I_{peak}}{V_{in}}\right) \\
&= \frac{2\ \mu H \times 0.417\ A}{2.4\ V} \\
&= 3.8\ \mu s
\end{aligned}
$$

$$
\begin{aligned}
Freq &= \frac{1}{T_{on} + T_d} \\
&= \frac{1}{T_{on} / 0.52} \\
&= 138\ kHz
\end{aligned}
$$

So, all we need is a square-wave signal at ~138 kHz to derive a logic power supply for the Stiquito.

Of course, a simpler solution is to use Maxim's wide-range regulated step-up converter chip—the MAX877. This chip can generate up to 200 mA at 5 V from a ~1.5–6.0-V power-supply range.

Also, the MOSFET and a variable PWM generator are integrated into this eight-pin chip. The MAX877 was designed for use in battery-operated computing devices like PDAs and portable phones.

GOING FURTHER

Hopefully, you've gotten some ideas for your next robot project. Robot power systems are a difficult issue. OK, perhaps not as bad as robot navigation, but pretty hard.

It should be interesting to watch the portable computing-device industry for battery-powered technology. Hopefully, we will see better battery technologies, as well as more efficient converters and chargers. It will be interesting to revisit this topic in a few years to see the changes.

If you're interested in power generation and storage, check out the references. In particular, *The Art of Electronics* has a whole chapter devoted to the topic of low-power techniques and a detailed discussion of primary and secondary battery types. The *Standard Handbook for Electrical Engineers* is also a great resource for power-related technologies. ■

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

REFERENCE

D.G Fink and H.W. Beaty, *Standard Handbook for Electrical Engineer*, McGraw-Hill, New York, NY, 1993.

D.G. Fink and D. Christiansen, *Electronics Engineers' Handbook*, McGraw-Hill, New York, NY, 1989.

P. Horowitz and W. Hill, *The Art of Electronics*, Cambridge Press, New York, NY, 1989.

D. Lines, *Building Power Supplies*, Master Publishing/Radio Shack, Ft. Worth, TX, 1991.

SOURCE

MAX877
Maxim Integrated Products
120 San Gabriel Dr.
Sunnyvale, CA 94086
(408) 737-7600
Fax: (408) 737-7194

I R S

404 Very Useful
405 Moderately Useful
406 Not Useful

MicroBot

FEATURE ARTICLE

Bruce Reynolds

Programming Intel's 8749 for Robotic Control

Bruce is an advocate of age-old wisdom: If you can do it simply and with common components, do it! As he proves with MicroBot, overkill is unnecessary. You can still cram a lot of functionality into an 8-bit micro.

From the first mechanical clunkers resembling a tin can with arms and legs to today's advanced techno-marvels deployed by NASA, robots have always captured our imaginations and heightened our anticipation for the future.

With the new advances in microprocessor technology and endless resources for technical information at our fingertips unleashed by the Internet, it's no longer a massive engineering feat to develop an experimental robotics platform on your own. The average designer now has the means to produce robotic creations with advanced capabilities.

The idea for MicroBot came about after a recent discussion with a not-so-technically-inclined friend in which he proudly announced that he thought of robots as simple boring machines.

My reaction: simple perhaps, boring never! Putting together a simple robot application is a great way to develop your engineering expertise.

To prove my point, I use MicroBot to acquaint you with Intel's 8749 micro and to demonstrate issues that need to be considered when you work with sequential control logic, servo control, timing, and power consumption.

I decided to base MicroBot, shown in Photo 1, on the Intel D8749H ceramic DIP 8-bit microcontroller. It's an older version of the 8-bit family from Intel, but it's still capable of handling many control applications.

The D8749H has 2K × 8 data EPROM, with 128 × 8 RAM, making it ideal for robots needing only small amounts of program memory. The ability to erase and reprogram the windowed version is handy when debugging assembly code. The 27 available I/O lines were more than enough for MicroBot's limited control requirements.

I chose the 3.57-MHz crystal because of its availability and because higher clock speeds aren't crucial to MicroBot's operation. If you want the timing routines written for MicroBot to work without having to modify them, stick with the 3.57-MHz crystal.

My idea was to build a small, simple robot from readily available parts and without using overkill tactics. I wanted a robot that could be programmed via onboard push-button switches to navigate through an obstacle course.

My goal: have the user program MicroBot to navigate the course in the shortest amount of time, thus winning the competition, gaining the respect of all present…and maybe, just maybe, having a little fun in the process.

OPERATION

Before getting into how I put this robot together, let me first tell you how I wanted MicroBot to operate.

If the user presses button 1 (Forward), the display shows the number 01. Pressing button 8 (Enter) clears the display, and the control bits for forward motion are recorded into the on-chip RAM.

Pressing button 6 (Time) displays a count from 1 to 60, which indicates the number of seconds MicroBot should proceed in the preselected direction. To stop the counting at the desired time, press Enter. The display digits update at approximately 0.5-s intervals.

Once Enter is pressed, the time data is stored in on-chip RAM and program control is then passed to the first routine (`Begin`, shown in Listing 1) to wait for more user entry.

Once all directions and times are entered, pressing the Run button causes MicroBot to execute the stored instruc-

tions for direction and time, thus attempting to navigate the obstacle course.

After executing the first programming sequence, MicroBot adds any further data entry to the end of the first stored sequence. You can add more to the stored direction and time data.

If you were close on the first programming attempt, you may be able to finish the obstacle course. If not, pushing Reset and Clear erases prior programming, enabling you to start over.

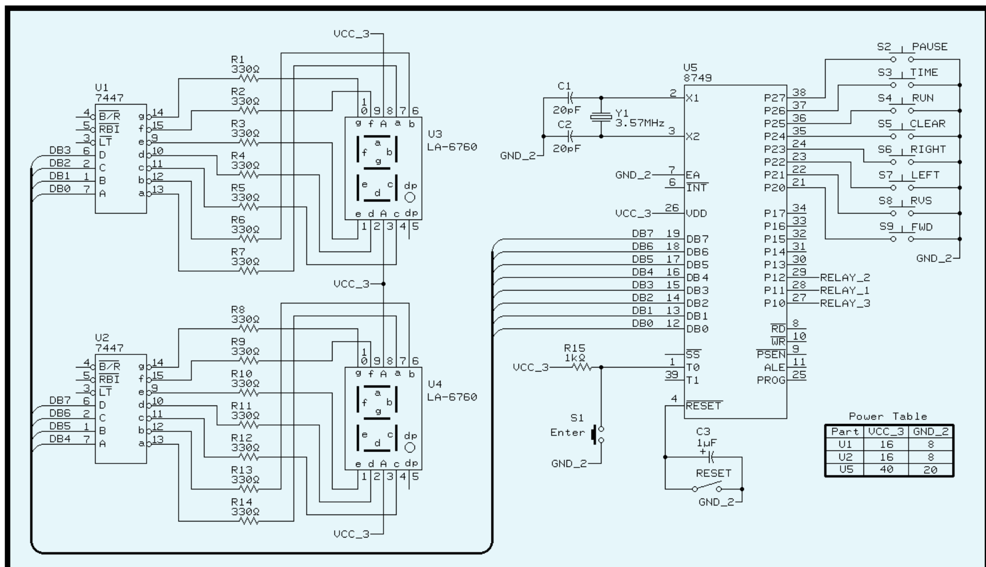


Figure 1—Pin 1 (T0) of the D8749H detects the Enter key input using the conditional transfer instruction `JT0`. *Port 2 keeps track of the rest of the key input. The LA-6760 seven-segment common anode displays verify user key presses and display the time count when setting the time. Port 1 uses only the three least significant bits to control the motor relays. The remaining five I/O pins can be used to add extra features.*

HARDWARE

As shown in Figure 1, port pins 12–19 (DB0–DB7) are used to output user data-entry information to the 7447 decoder/drivers that drive the LA-6760 seven-segment displays. Port 2 handles the user input of forward, reverse, left, right, pause, time, clear, and run. Port 1 uses the three least significant bits to control three Omron G5V-2 DPDT relays, which control motor power and direction.

The switch connected to pin 1 (T0) is for the Enter key, and the conditional transfer instruction `JT0` detects when Enter is pressed. All instructional function keys (i.e., forward, reverse, left, right, pause, and time) wait for the Enter key to be pressed before returning program control to `Begin` and waiting for more user data entry.

Figure 2 illustrates the simple control scheme for the motors, using relay 3 as the power-on/-off control to the motors. Relays 1 and 2 simply reverse polarity to each motor to provide forward and reverse motion.

The motors I selected for MicroBot are modified Futaba FP-S148 servos. They provide an output torque of 42 oz./in. (3 kg/cm) to power through rough terrain and weigh a mere 1.5 oz each. Subsequently, they are lightweight and powerful enough to be used in many robotics applications.

In their unmodified state, pulse-proportional servos are designed for use in radio-controlled cars and planes. They require control pulses from 1 to 2 ms long, repeated 60 times per second.

The servo positions its output shaft in proportion to the width of the pulse. A 1.5-ms pulse centers the shaft. A 1-ms pulse positions the shaft to the left 45°, and a 2-ms pulse moves the shaft to the right 45°. Standard unmodified servos are an exceptional choice for precision positioning applications. Since MicroBot requires a full 360° rotation of the wheels, the servos have to be modified prior to use.

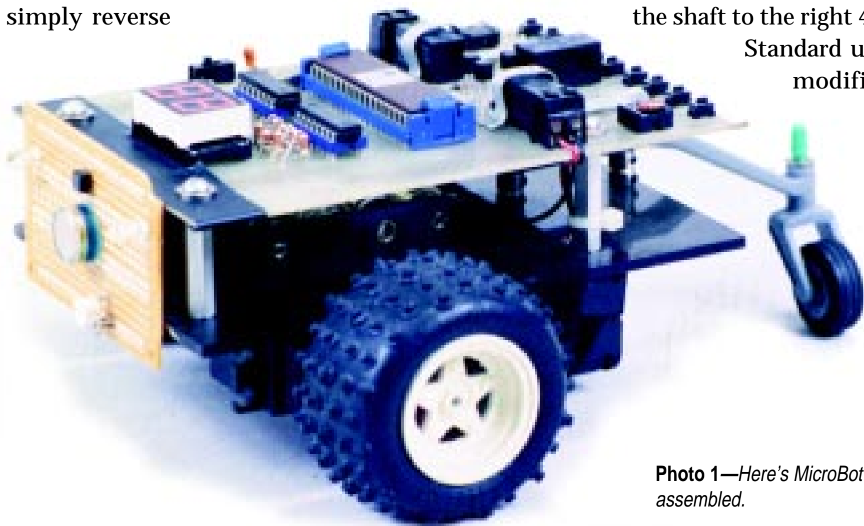


Photo 1—*Here's MicroBot fully assembled.*

To modify these servos, just remove the drive electronics from inside the servo case and cut the nib off the final gear. Next, remove the three wires from the circuit board and solder the red and black power wires directly to the motor power tabs. The white (control) wire may be discarded, because no positioning pulses are needed.

Next, replace the motor and reduction gears. Modifying the servo in this way makes it possible to use simple relay or digital control techniques with the servo and achieve the full 360° of rotation.

Since no two servos ever seem to be created equal, using variable resistors or experimenting a little with fixed values provides some equalization of speeds between the two motors. If you notice that MicroBot tends to veer slightly while moving forward, you can adjust individual motor speeds by adding or subtracting resistance values.

POWER SUPPLY

Figure 3 shows the individual power-supply sections. As a general rule of thumb, it's good practice to include some type of regulation in any circuit you design. However, when cost and a

low parts count are important factors (and efficiency isn't), it can sometimes be avoided for the more simple circuits.

The MicroBot power supply was designed to use AA batteries without regulation. Since a linear regulator like the LM7805 requires an input voltage of 7.0 V or higher to maintain a regulated output of 5.0 V, larger and heavier batteries are required, adding to the weight, parts count, and cost. Linear-regulator power loss also adds to circuit power consumption. For battery-operated platforms, you want to avoid any unnecessary power loss.

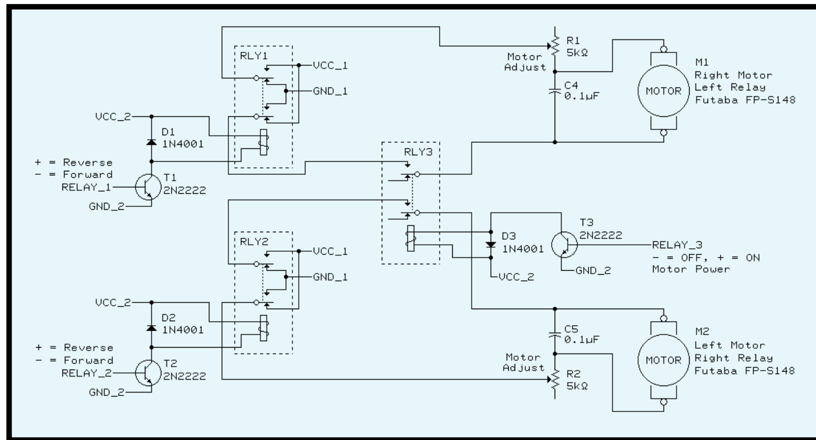


Figure 2—*This relay setup controls MicroBot's servos. Relay 3 serves as a power switch. Relays 1 and 2 reverse polarity to each motor, providing forward and reverse motion. When power is applied, relay 3 turns off to stop both motors. Relays 1 and 2 in the normal off state supply power to the motors for forward motion. When attaching the motors to the relay outputs, the left motor connects to relay 2 and the right motor to relay 1. The 0.1-µF capacitors are soldered across the servo power leads inside the servo case.*

The 6.0-V supply for the servo motor section consists of four 1.5-V alkaline AA batteries. Each servo draws approximately 70 mA with a 6.0-V supply, for a total power consumption of 140 mA. Good-quality alkaline batteries normally provide ~1.5 h of motor operation. Using separate 6.0-V supplies for each motor could extend this time, but not without a tradeoff of increased weight and load on the motors.

Power to the microcontroller and relay section comes from eight 1.5-V alkaline AA batteries in series. Ground is tapped between the fourth battery, which is a common ground for the microcontroller and relay circuit.

The Omron G5V-2 relays are rated from 5 to 6 V and consume about 60 mA each with a 6-V supply. The motors require only the activation of relay 3 for forward motion using 60 mA.

Since MicroBot normally covers more ground in the forward direction, this configuration ultimately saves on power consumption. When MicroBot is going in reverse, the relay section consumes 180 mA, and a left or right turn requires 120 mA. While this is indeed a simple process, it is important to remember this during design.

The D8749H microcontroller's absolute maximum voltage rating is 7.0 V [1]. Operating at 6.0 V keeps the project within this limitation. Here again, the flexibility of the D8749H comes into play. Power consumption for the microcontroller section is ~195 mA without the seven-segment displays active and 290 mA with the displays active during user programming.

Figure 4 depicts an optional light-sensitive headlight assembly. By adjusting the variable resistor, you can select the level of darkness required to activate the headlights. I used garden-variety (super bright) red LEDs, which are quite effective with a distance of about 6′.

CONTROL CODE

The control software was kept extremely simple, yet it very effectively controls MicroBot. With a little creativity at the keypad, one can make MicroBot seem quite lifelike and even appear to be making intelligent decisions about its environment.

The code begins by selecting register bank 0, establishing Direction and Time RAM location pointers. It then sends data to clear the display, stop the motors, and wait for key entry on powerup and on return from other key-entry routines. Registers R0 and R1 are indirect address pointers for direction and time storage locations, respectively.

MicroBot uses 104 internal RAM storage locations starting at location 24d, just above the eight-level stack, up to 127d, allowing for up to 52 directions with 52 corresponding time periods for storage and execution.

The first version of MicroBot was assembled on a breadboard. But, due to the somewhat overzealous contest

Listing 1—*The beginning code segment sets up direction and time RAM pointers, clears the display, halts the motor, and waits for user key entry. You can get the remaining code segments from the Circuit Cellar Web site.*

```
        org   0         ; Start at 0
        sel   rb0       ; Select register bank 0
        mov   r0,#18h   ; Set DIRECTION RAM location pointer
        mov   r1,#19h   ; Set TIME RAM location pointer

begin:  mov   a,#0ffh   ; Load clear display bits
        outl  bus,a     ; Clear display
        mov   a,#0f0h   ; Bits to halt all motors (CLR P1.0)
        outl  p1,a      ; Halt motors
        clr   a         ; Clear accumulator
        in    a,p2      ; Get user keypad input
        cpl   a         ; Invert keypad entry 0 = 1
        jb0   fwd       ; If Acc Bit 0 = 1 goto fwd
        jb1   rvs       ; If Acc Bit 1 = 1 goto rvs
        jb2   left      ; If Acc Bit 2 = 1 goto left
        jb3   right     ; If Acc Bit 3 = 1 goto right
        jb4   clear     ; If Acc Bit 4 = 1 goto clear/clear ram
        jb5   run       ; If Acc Bit 7 = 1 goto run/execute pgm
        jb6   time      ; If Acc Bit 6 = 1 goto time/set time
        jb7   pause     ; Pause/Stop routine
        jmp   begin     ; Recycle until keypress detected
```

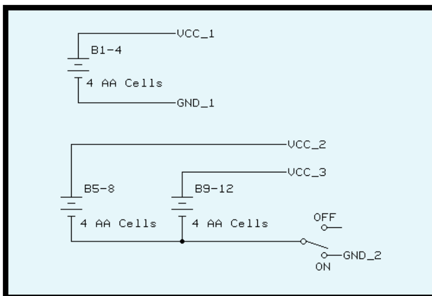


Figure 3—*MicroBot's power supply consists of 12 AA alkaline batteries. To simplify battery placement and connections, three battery holders with four AAs in each are used to house the batteries.*

participants, the breadboard construction method seemed weak at best. Wires and wheels were soon flying about as excited, would-be contestants crowded around grabbing at MicroBot.

Soon MicroBot and I were back in the lab, so I could add a little armor plating. The final version is on a single-sided, circuit board attached to Plexiglas via standoffs, allowing placement of the motor batteries and a place to secure the servos with hot glue.

The rear (center) wheel is a model-airplane tail wheel assembly. The rubber wheels came from a remote-controlled car that didn't survive a day after Christmas, and they're attached to the servo horns with epoxy.

SIMPLE, NOT LIMITED

Simplicity is a breath of fresh air. It's amazing what you can create with common parts and a single simple micro.

With the ever-advancing onslaught of new and improved, super-charged 16-bit micros, it's easy to feel hard pressed about deciding which device would prove most efficient for a particular application. But remember, there's still a place for simple 8-bit and even 4-bit microcontrollers [2].

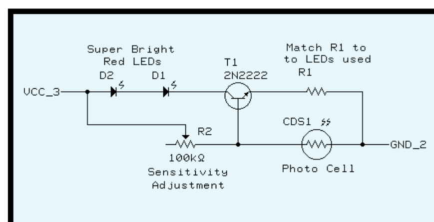


Figure 4—*In the optional headlight assembly, the Radio Shack super-bright red LEDs have a laser-like quality and cast a narrow spot ~6′ in front of MicroBot. For maximum effect, keep the value of R1 as low as possible.*

I am often approached by beginning microcontroller designers with the same questions: "What is the best device type to use?" and/or "What is the most efficient assembler?"

Without the concerns of going to full production with a new product or the cost of large volume manufacturing, I recommend experimenting with as many device types as possible—even older 8- and 4-bit low-end models. (If you're looking for how-tos, check out *Mobile Robots* [3].)

This experience helps build a well-rounded knowledge of device capabilities and design techniques. Break new ground, and don't limit yourself to one device type. Bigger and faster is not always the answer. ◼

Bruce Reynolds works for the Colorado State Department of Corrections as an electronics supervisor. He also operates Reynolds Electronics, providing contract engineering services for 8051-based embedded control systems, as well as building and consulting for new computer systems. You may reach Bruce at breyno@rmi.net.

SOFTWARE

Complete source code for this article is available via the Circuit Cellar Web site.

REFERENCE

[1] Intel, Publication 270646-005, 1993.

[2] S. Ciarcia, "A Computer-Controlled Tank," *BYTE*, 6:2, 80–93, 1981.

[3] J.L. Jones and A.M. Flynn, *Mobile Robots*, A.K. Peters, 1993.

SOURCE

D8749H 8-bit microcontroller
Intel Corp.
5000 W. Chandler Blvd.
Chandler, AZ 85226-3699
(602) 554-8080
Fax: (602) 554-7436
www.intel.com

I R S

407 Very Useful
408 Moderately Useful
409 Not Useful

36 **Nouveau PC**
edited by Harv Weiner

41 **Converting 8051 Code for an *'x86* Embedded Processor**
Chip Freitag & Jeff Kirk

47 **Real-Time PC**
Picking a PC RTOS
Ingo Cyliax

53 **Applied PCs**
Embedding PC Card
Part 1: The Time Has Come
Fred Eady

Photo courtesy of
Advanced Micro Devices, Inc.

NPC

FLASH MEMORY

ZF MicroSystems is offering flash-memory chipsets certified to work with embedded systems based on the company's Single-Device PC (SDPC) OEModules.

The new **ZF FlashDisk-SC** chipsets consist of flash-memory and controller devices, which are guaranteed by the company to work with the recently announced SMX/386-40 OEModule and other ZF MicroSystems products. The SMX/386-40 is a complete 40-MHz, '386-based PC in a device that measures approximately 2″ × 3″, complete with ISA bus, floppy and hard drive controllers, system memory, and serial and parallel ports.

The ZF FlashDisk-SC, which is available in 2-, 4-, 8-, and 12-MB versions, offers high performance for a lower price than other flash-memory chipsets in the industry. The full-boot operability and superior read/write speeds make this chipset ideal for high-performance demands in embedded-systems design. The chipset offers full read/write disk emulation and contains ECC/EDC for high data reliability. The chipsets are an industry-standard design and provide an easy-to-use interface.

The ZF FlashDisk-SC is available now and is sold together with the SMX-386 OEModule. Prices start at **$40** in quantities of 100.

ZF MicroSystems
1052 Elwell Ct.
Palo Alto, CA 94303
(415) 965-3800
Fax: (415) 965-4050
info@zfmicro.com • www.zfmicro.com

#510

ISDN ACCESS IN PC/104 FORMAT

Xecom has announced a PC/104-card family for connecting to the Integrated Services Digital Network (ISDN). Six boards offer various combinations of ISDN, analog modem, RS-232, and POTS interfaces. Two high-speed serial ports with 16C550 UARTs provide interface to the onboard ISA-bus connector. Applications include remote data collection and transaction processing, process monitoring and control, network monitoring, audio transmission, and remote LAN and Internet/Intranet access.

The **PCISDNU**, a single-board ISDN terminal adapter with integral NT1 (U interface), provides a serial data channel capable of 1200 to 64,000 bps synchronous or 300 to 115,300 bps asynchronous over one ISDN B-channel. The ISDN interface includes the line termination with all passive components. Only a standard two-wire phone cable is needed for hookup to the ISDN wall jack, and the firmware is compatible with all standard ISDN central-office switches used in North America.

The **PCISDNUP** lets you connect an analog telephone, modem, or fax machine directly to the PC/104 board. It provides dial tone, ring voltage, and calling progress tones to the POTS line, so it uses the full potential of the high-speed ISDN interface without requiring ISDN-compatible telephone or fax equipment.

Other products combine ISDN terminal-adapter and analog-modem functions at 14.4-, 28.8-, or 33.6-kbps data rates. The modem runs on a second high-speed RS-232 channel, providing an analog back-up line when ISDN service isn't available or acting as a lower speed data line. Serial channels are jumper configurable to COM ports 1–4 and IRQs 3–4. The boards are in a 90 mm × 96 mm × 15 mm (3.6″ × 3.8″ × 0.6″) configuration with stackthrough pins.

Single-quantity pricing for the PCISDNU is **$339**.

Xecom, Inc.
374 Turquoise St.
Milpitas, CA 95035
(408) 945-6640
Fax: (408) 942-1346
www.xecom.com

#511

Nouveau PC

edited by Harv Weiner

EMBEDDED PROCESSOR MODULE

Intel's **Embedded Processor Module** is a high-performance subsystem for embedded, industrial, and communication applications where flexibility and the ability to upgrade are important. The module contains the 133-MHz mobile Pentium processor or the 166-MHz Pentium MMX, 82439HX system controller, 256-KB pipeline-burst SRAM L2 cache, clock generator, and a voltage regulator for the Pentium processor.

The module consists of a six-layer board fabricated with FR4 laminate with top and bottom signal layers, separate power and ground layers, and two internal signal layers. It measures 3″ × 4″, double sided with two high-performance low-profile connectors and a heat sink for the Pentium processor.

A development kit, available from VenturCom, includes the Embedded Processor Module, evaluation board, interposer board for voltage and power measurement, and schematics.

Software includes evaluation copies of the QNX RTOS and Photon microGUI windowing system with Watcom C/C++ compiler and tools, Cogent Slang programming language for QNX and Photon microGUI, RadiSys Intime with Intrinsyc Integration Expert, VenturCom RTX with Component Integrator, and PhoenixPICO BIOS.

Pricing is **$400** in single quantities.

Intel Corp.
2200 Mission College Blvd.
Santa Clara, CA 95052-8119
(408) 765-8080
www.intel.com/design/intarch **#512**

MOBILE PC

The **PC-500** packs high-performance features into 5.75″ × 8″, including a 133-MHz 5x86 CPU, five serial ports, 10BaseT Ethernet, SCSI-2 interface, advanced flat-panel video controller, and 24 lines of bit-programmable DIO. It withstands 20 G of shock, 3 G of vibration, and operation from –40° to 70°C at full CPU speed. It operates stand alone or with PC/104 expansion.

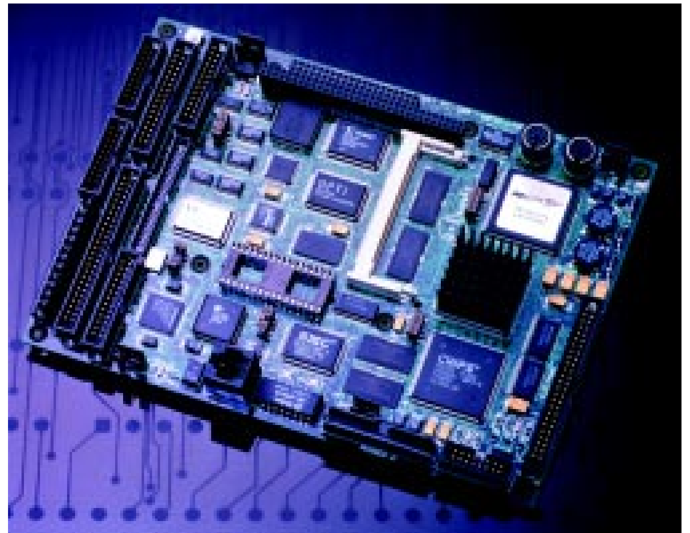
The card also features 2-MB flash memory with resident DOS 6.22, 1–33-MB EDO DRAM, and real-time video with graphics accelerator. It has 2-MB video RAM; IEEE 1284 multifunctional parallel port; floppy and hard drive interfaces; keyboard, speaker and mouse ports; and a watchdog timer. And, the card offers a real-time clock and optoisolated interrupts.

The PC-500 supports leading OSs like Windows NT and QNX. The on-card flash contains DOS 6.22 and diagnostic software to test and verify on-card I/O and memory functions. Application programs can be stored in the resident flash memory or on an external 24-MB flash card, eliminating the need for a hard drive.

The card supports CRTs and LCD, plasma, and EL flat-panel displays. Since the video circuitry operates on the Local bus at full processor speed, high-performance programs execute rapidly. The video RAM supports high-resolution displays to 1024 × 768.

The PC-500 costs **$995** and sells for less than **$700** in OEM quantities.

Octagon Systems
6510 W 91st Ave.
Westminster, CO 80030
(303) 430-1500 • Fax: (303) 412-2050
www.octa.com **#513**



NouveauPC

RUGGED CompactPCI ENCLOSURE

The **PC/Ranger** is a rugged enclosure for 3U CompactPCI boards. The Nema 4-sealed PC/Ranger is designed for use in high-vibration, high-shock, and exposed environments such as trains, aircraft, trucks, military vehicles, and outdoor areas.

In the PC/Ranger, off-the-shelf CompactPCI boards are individually mounted into shock-absorbing card guides and locked in place with retaining screws. Off-the-shelf CPU boards are cooled with a supplied companion conduction cooling card to remove heat from Pentium-class CPUs to the chassis. The PC/Ranger has a mounting flange to dissipate heat by conduction to a cold plate as well as cooling fins to dissipate heat by convection to the ambient air.

Pricing for the PC/Ranger starts at **$1895**.

Kinetic Computer
76 Treble Cove Rd.
Billerica, MA 01862
(978) 439-0500
Fax: (978) 439-0501
www.kin.com **#514**

NouveauPC

Chip Freitag &
Jeff Kirk

Converting 8051 Code for an 'x86 Embedded Processor

Although many prefer using C when moving to a 16-bit processor, there are times when low-level drivers and time-critical code demand that portions of the code remain in assembler. Chip and Jeff suggest ways to ease that process.

Perhaps you're running out of memory space with the 8051, needing more performance, or designing a totally new product. You're convinced you need to switch architectures, and you've looked at the various options available.

Maybe you're leaning towards the Am186 family of processors, but you have many man-years invested in assembly-language code and are dreading the thought of throwing it all away and starting over. You need to know how difficult it is to write 80186 assembly-language code.

The good news is that it's reasonably easy to migrate—both from a hardware and a software perspective. We'll show you how.

In this article, we're assuming you're an experienced 8051 user but haven't used the Am186/188 before. Just as a quick overview, the Am186 family of embedded microprocessors is 100% instruction-set compatible with the Intel 80186. It includes standard 80186 peripherals such as timers and memory/peripheral chip selects.

The Am186 family also offers integration like async and sync serial ports, programmable I/O, and 32 KB of SRAM. Speed grades of up to 40 MHz are available.

REGISTERS

The 8051 microcontroller is somewhat unique in that its special-function registers are located in RAM. This setup is possible because the 8051 was designed with a small amount of SRAM on the die, so there's no speed penalty for accessing registers in RAM.

Since you can access the Accumulator (A) as both a register and direct memory location, you can do things like add the accumulator to itself (e.g., `ADD A, acc ;` where `acc` = E0h).

The register set of the 80186 processor follows a microprocessor model rather than a microcontroller model. Usually, there's no on-chip RAM on 80186 processors.

Perhaps the biggest consequence of this difference is the lack of banked indexed registers (R0–R7) on the Am186. Otherwise, the 80186 register set (shown in Figure 1) is functionally similar to the 8051's.

In general, the 80186 register set is 16 bit rather than 8, but some registers (i.e., AX, BX, CX, DX) are also byte addressable

8051 Register	AM186 Register	Function
ACC(A)	AX	Accumulator
B	DX	Auxiliary accumulator & multiply results
PSW	FLAGS	Processor status flags
SP	SP	Stack pointer
DPTR	BX	Base index register
R0–R7	DI,SI	Index registers

Table 1—This table shows the approximate register-set equivalents between the 8051 and 80186 processor families. The 8051 does have more index registers. The 80186 AX register can be treated as two separate eight-bit registers.

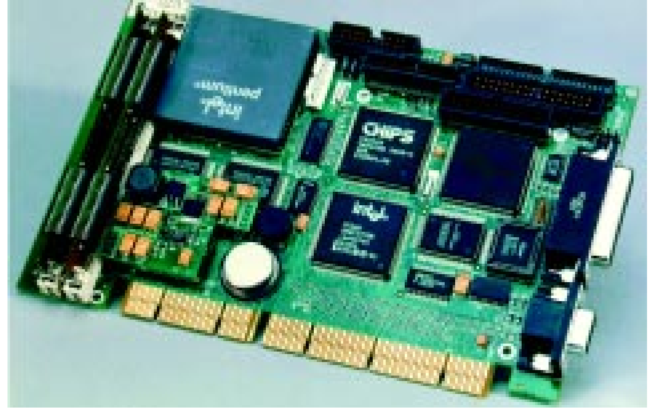
NPC

PENTIUM MMX SBC

The **2107** from Toronto MicroElectronics is the first half-sized (4.8″ × 7.8″) industrial SBC that supports Intel Pentium MMX and AMD k6 processors at speeds up to 266 MHz. Features include L2 pipeline-burst cache, complete standard I/O (i.e., two serial ports, one parallel port, floppy and EIDE interfaces), up to 256-MB FPM or EDO DRAM on two 72-pin SIMM sockets, and PCI and ISA buses on a passive backplane, with a very small form factor.

The board features a flat-panel display interface using C&T 65548 with 1-MB display memory. Its LVDS (low-voltage differential signal) drives the flat-panel display cable up to 100′. It also minimizes EMI, which helps the system designer meet FCC, DOC, CE, or other regulatory requirements.The 2107 supports most flat-panel displays (e.g.,TFT, passive LCD, gas plasma, FL, etc.).

Embedded-PC system features include a watchdog timer with software or hardware disable/enable, 128 bytes of EEPROM for system parameters, up to 384 KB of EEPROM for user system parameters, real-time clock, fully AT-compatible BIOS, power-failure detection circuitry, and PC/104-bus capability.



Toronto MicroElectronics, Inc.
5149 Bradco Blvd. • Mississauga, ON
Canada L4W 2A6
(905) 625-3203 • Fax: (905) 625-3717
www.tme-inc.com **#515**

NouveauPC

and can function much like the 8-bit 8051 versions.

Table 1 has a short list of the core 8051 special-function registers with the 80186 equivalents.

MEMORY SPACE AND ADDRESSING

The 8051 and 80186 have different memory schemes, but in many ways, they're very similar.

The 8051 divides memory into two categories—on-chip and external. External memory is subdivided into program and data memory, which enables the 8051 to double the 64-KB address space of a standard 16-bit address.

On the other hand, the 80186 has no on-chip address space and divides external memory into two categories—memory and I/O. I/O space has its own set of dedicated instructions.

Each of these two schemes has an effect on both instructions and addressing modes. For example, the 8051 forces you to access external memory indirectly through the data pointer (`DPTR`). This limitation is such a problem that some 8051 versions add a second data pointer to try to ease this bottleneck.

The 80186 has no such limitation (you can address external memory and I/O directly). However, if indirect addressing is desirable, it can be done with one of the BP, BX, DI, or SI registers.

In general, the 80186 has more types of addressing modes that are more powerful than the 8051. Table 2 gives the approximate 80186 equivalents of the standard 8051 addressing modes, but it's worthwhile investigating the more powerful modes unique to the 80186.

One last addressing topic deserves brief discussion: the 80186 is a segmented processor. This concept should be easy for 8051 users to understand. Think of the 80186's memory space as a collection of 8051-sized code and memory spaces, or in short, one segment equals an 8051 64-KB memory space.

Each segment register points to one of these 64-KB spaces. The memory's physical address is generated by shifting the 16-bit segment address to the left four bits (multiplying by 16) and adding it to the 16-bit offset. The result is a 20-bit address that reaches a 1-MB address space.

Consequently, the 64-KB spaces can overlap, which is useful in smaller systems with limited memory. The segment registers are:

- DS (data segment)—holds data, like the 8051 external data memory
- CS (code segment)—serves as default locations for instructions, like 8051 program memory
- SS (stack segment)—is the location of the stack, like 8051 internal stack space
- ES (extra segment)—acts as a spare, often used for string operations

As you see, you can think of the 8051 as a segmented processor with three types of segments (internal, data, and program) consisting of a single segment each.

The 80186 has a few addressing modes that the 8051 doesn't. In most cases, these addressing modes aren't needed and can be ignored.

But, you can understand the different addressing-mode possibilities better if we separate them into three categories—data (`MOV`, `AND`, etc.), program (`CALL`, `JMP`), and stack (`PUSH`, `POP`).

DATA ADDRESSING MODES

As we mentioned, the physical address consists of segment plus offset. The seg-

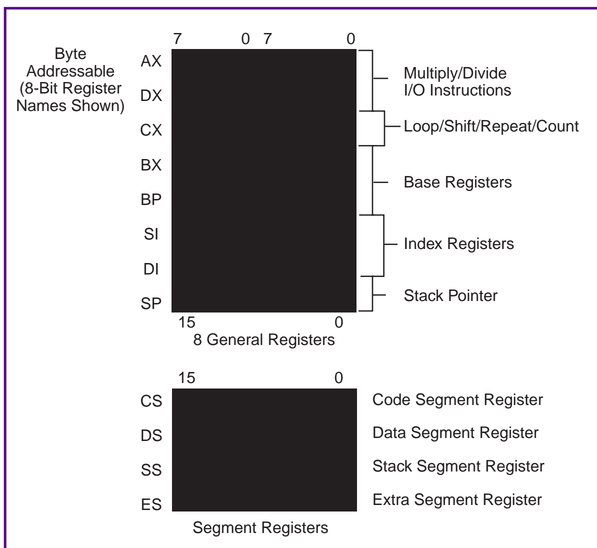


Figure 1—The 80186 microprocessor family register set includes several more general-purpose registers than the 8051. This setup allows for more efficient code as operands don't have to be temporarily saved to free up the accumulator.

ment register is usually implicitly chosen by the addressing mode but can be explicitly chosen with a segment override prefix.

On the other hand, the offset can be composed by summing one or more of three address elements:

- displacement (D)—an 8- or 16-bit immediate value contained in the instruction
- base (B)—contents of the BX or BP base registers
- index (I)—contents of the SI or DI index registers

These three elements are combined into the six data-addressing modes given in Table 3. Unfortunately, not all assemblers use the same notation. In general, there are some minor differences between the common 8051 and 80186 syntaxes.

STACK AND PROGRAM ADDRESSING

The addressing modes of these two groups are about the same between the two processors with a few differences. We discuss the most important ones here.

The stack-addressing instructions (i.e., `PUSH` and `POP`) are almost identical. Perhaps the biggest difference is that these instructions use the Stack Segment register by default. The location of the stack segment is usually affected by the "model" (an assembler directive), so consult your assembler's user manual.

Program addressing has a few more differences. There is no equivalent of the 11-bit addressing modes of the 8051 (`AJMP` and `ACALL`). Otherwise, they both support direct, relative, and indirect addressing for program branching.

The 8051 only supports one instruction for indirect program branching (i.e., `JMP @A+DPTR`), while the 80186 is a lot more flexible, including the capability to do a double indirect jump or call. This feature can be useful for program structures such as jump tables.

ADDRESSING-MODE SUMMARY

The 8051 is essentially a subset of the 80186. If you only need the capabilities of the 8051, it's possible to keep your code fairly simple.

On the other hand, the more powerful capabilities of the 80186 make a lot of tasks easier. Here are a few simple hints about addressing.

First of all, the operand field often determines the size of the transfer (`AX` vs. `AL`). Also, either the source or destination must be a register (no memory to memory), with the exception of some string operations.

As well, don't mix data sizes (e.g., `mov AX,CL`). And finally, remember that in most cases, the segment register is implied but can be overridden (defaults are BX, DI, and SI equal to data, and BP equal to stack).

ARITHMETIC OPERATIONS

The 8051 arithmetic instructions are a subset of the arithmetic operations the 80186 can perform. The 80186 can perform 8- or 16-bit arithmetic operations, so it's fairly easy to port 8051 code using the 8-bit operations.

Naturally, floating-point code is a lot faster using the 16-bit math operations. New 80186 instructions include signed multiplication and division and several ASCII adjust instructions.

LOGICAL OPERATIONS

Like the arithmetic operations, the logical operations of the 8051 are a subset of the 80186. The logical operations can be 8 or 16 bit. Unlike the 8051, which has only single-bit rotate instructions, the 80186 allows multiple-bit shifts and rotates using `CL` or an immediate byte to specify the number of shifts to perform.

Notably, many operations restricted to the 8051's accumulator (`CPL`, `RR`, etc.) are open to any 80186 register or memory.

The 80186 also has a new set of arithmetic shift operations. These instructions can perform 8- or 16-bit shifts in either direction and include the carry bit in the shift operation.

DATA TRANSFER

The data-transfer capabilities of the 80186 are almost a superset of the 8051. The only unique 8051 instruction is `XCHD`, which requires several 80186 instructions to perform.

Be aware that most 80186 data-transfer operations are less restrictive than the equivalent 8051 instructions. For example, `MOVC` can only read a byte from code space. The equivalent 80186 operation (moving a byte, word or string from/to the code segment) has no such restriction.

8051 Mode	80186 Mode	Addressing Function
Rn	Register	Register addressing (register holds data)
direct	Direct	Direct memory address (memory holds data)
@Ri	Register Indirect	Indirect address (register holds address)
#data	Immediate8	8-bit constant included in instruction (immediate)
#data 16	Immediate16	16-bit constant included in instruction (immediate)
addr 16	Direct (& Far Dir)	16-bit destination (LCALL/LJMP form of #data)
addr 11	(none)	11-bit destination(ACALL/AJMP form of #data)
rel	Displacement	PC relative (short jumps)
bit	(none)	Direct memory address of a bit

Table 2—There is a good correspondence between the addressing modes of the 8051 and 80186. The only real mismatch is the bitwise addressing mode of the 8051, which is typically reproduced with a read-mask-compare 80186 sequence.

80186 Mode	Offset Calculation	Example
Register	(none)	Mov ax,bx
Immediate	(none)	Mov ax,#0
Direct	D	Mov ax,ds:4
Register Indirect	B or I	Mov ax,[si]
Based	B + D	Mov ax,[bx]4
Indexed	I + D	Mov ax,[si]4
Based Indexed	B + I	Mov ax,[si][bx]
Based Indexed with Displacement	B + I + D	Mov ax,[si][bx]4

Table 3—The 80186 data addressing modes provide efficient access to high-level data structures. This table also shows examples of typical assembler syntax.

The 80186 adds several new data-movement instructions, which include instructions to push and pop the flags to the stack and push and pop all registers. The new 80186 instructions also include the `IN` and `OUT` instructions, which operate on the separate I/O space via peripheral chip-select pins, reflecting this difference in the architecture of the two processors.

PROGRAM CONTROL

At first glance, it looks like the 80186 doesn't cover all of the 8051's branching instructions. On closer examination, however, we find that the 80186 has close equivalents in all cases.

The 8051 is a little more flexible about which register and memory can be used as a loop counter, but the 80186 has more sophisticated ways of terminating a loop (i.e., count or comparison).

The 80186 provides many conditional jump instructions, allowing jumps on the value of most flag register bits. Besides the short relative jumps and calls the 8051 provides, 11-bit absolute jumps and relative "near" jumps and calls, as well as segment-plus-offset "far" jumps and calls are supported.

One subtle—but very important—difference between the two processors is the `JZ` instruction. On the 8051, this instruction branches if the accumulator is zero, but on the 80186, the branch is taken if the ZF flag is set.

The `JCXZ` instruction tests the CX register for zero, so it could be used if you don't want to add an extra compare with zero. The `LOOP` instruction is essentially the same as `DJNZ`, the only difference being that `LOOP` is restricted to using CX as its counter, while `DJNE` can use any register or direct byte.

`LOOPE` and `LOOPNE` do the same thing as `LOOP` as well as examining the ZF flag. These instructions are usually combined with either `CMP` or `TEST` to properly set the ZF flag. For example, you can search through a fixed-length string for a specific pattern of set bits by putting the length of the string in `CX` and using the `TEST` instruction.

BOOLEAN DATA MANIPULATION

The biggest limitation of the 80186, as compared to the 8051, is the lack of bit addressing. Without bit addressing, many of the 8051 Boolean instructions have no 80186 equivalents. However, all of the missing instructions can be simulated with a small number of 80186 instructions.

In general, the 80186 can set, clear, or complement only the carry bit. Most other bits must be either masked for and tested explicitly or somehow shifted to the carry bit.

One common application is to set or clear port bits (PIOs). Listing 1 shows how to do this on a 80186-family microcontroller. The 8051 can do this type of operation with one instruction.

STRING OPERATIONS

Among the nice features of the 80186 family are the string instructions, which move string data between registers, memory, and/or I/O space. Automatic comparison and scanning can also be performed.

The `CLD` and `STD` opcodes enable the direction of the string movement to be

Listing 1—Setting a programmable I/O bit on an Am186 microprocessor involves reading the PIO data register, masking for the specific bit (or bits), and then writing the result back out the data register.

```
mov   dx,PDATA1   ;point to PIO1 DATA register
in    ax,dx       ;read current value
or    ax,0x0040   ;set PIO bit 6 (to make it high)
out   dx,ax       ;write changed value back to the port
```

Listing 2—This listing shows an example of moving a block of memory on the 8051. The availability of only one data pointer makes this an awkward chore, since the source and destination addresses must be swapped twice for each iteration of the loop.

```
  MOV  src_h,#20h   ;initialize high byte of source pointer
  MOV  src_l,#00h   ;initialize low byte of source pointer
  MOV  des_h,#40h   ;initialize high byte of destination pointer
  MOV  des_l,#00h   ;initialize low byte of destination pointer
  MOV  R0,#57h      ;initialize block length

top:
  MOV  DPH,src_h    ;get source pointer
  MOV  DPL,src_l
  MOV  A,@DPTR      ;get byte from source block
  INC  DPTR         ;prepare for next source byte
  MOV  src_h,DPH    ;save source pointer
  MOV  src_l,DPL
  MOV  DPH,des_h    ;get destination pointer
  MOV  DPL,des_l
  MOV  @DPTR,A      ;write byte to destination block
  INC  DPTR         ;prepare for next destination byte
  MOV  des_h,DPH    ;save source pointer
  MOV  des_l,DPL
  DJNZ R0,top       ;decrement count and branch
```

controlled. CLD clears the direction flag, enabling the index registers to increment after the operation. STD allows the index registers to decrement.

Each string instruction operates on a single component—byte or word—of a string. Combining the string instructions with repeat prefixes enables multiple byte and word operations. Prefixes aren't really instructions. They assemble as part of the repeated string instruction and only operate on a single instruction.

The REP MOVS instruction is particularly useful for block memory transfers, which are always a problem on the original 8051 since it has only one data pointer (DPTR). A typical block transfer on the 8051 usually looks like Listing 2.

The situation is slightly more complicated when moving data from program memory to RAM since the only available instruction is MOV A,@A+DPTR (the accumulator needs to be reloaded each cycle). Listing 3 shows the equivalent operation on the 80186. Obviously, the 80186 code is much easier to read.

OTHER INSTRUCTIONS

There are a variety of new complex instructions in the 80186 instruction set, including the all-important CLI and STI, which disable and enable maskable interrupts.

Check out the XLAT instruction, which can be useful in embedded systems for table-lookup tasks like converting BCD to seven-segment LED. The BCD value in AL is used to look up the seven-segment value from a table in memory, and AL receives the new value.

As another example, the 80186 instruction set includes several instructions (e.g., ENTER, LEAVE, and BOUND) that can be used by a compiler to efficiently implement higher level languages (e.g., C or C++).

Other examples include the LOCK instruction, which can prevent external bus masters (as well as the internal DMA) from interrupting nonatomic events like repeated string operations. It's unlikely that converted 8051 code will need to use most of these classically CISC instructions, but it's good to understand what's available.

Listing 3—In contrast to the 8051, the 80186 provides specific instructions for moving blocks of data. This elegant code example takes advantage of these instructions, specific source and destination pointer registers, and a counter register to perform block moves of up to 64 KB.

```
MOV   SI,2000h    ;load source pointer (immediate addressing)
MOV   DI,4000h    ;load destination pointer
MOV   CX,57h      ;initialize block length
CLD
REP   MOVSB       ;move byte string
```

EPC

ON YOUR OWN

To aid programmers in converting code from the 8051 to the 80186, there is a Perl script that performs the basic conversions derived from this article. It isn't a turn-key conversion program, since it can't account for the inevitable system-dependent cases.

However, this program can at least be run against your source code and do a lot of the work for you. With a little knowledge of Perl (which might hurt at first, but will be good for you in the long run), it can be tailored to get you most of the way there.

For more information on code conversion, check the References. Subbarao focuses mostly on older 16-bit processors instead of the newer 32-bit versions, and his book is well-suited to embedded applications.

Brey covers the entire 8086 family, so he gives a lot more information on the 32-bit processors up through Pentium Pro. However, chapters 3–6 give an excellent description of the instruction set and addressing modes of the 8086.

This book also covers many of the notational differences between the various 80186 assemblers, so it's a good choice if you'll eventually need to move up to the '386 or better.

We hope you have enough information now to feel comfortable converting 8051 code for an '*x*86 processor. Even though the two processors were designed with different philosophies, they're surprisingly similar.

As you've seen, the 8051 is largely a subset of the 80186. With a little forethought, it should be easy to port 8051 assembler code to any of the Am186 family of microcontrollers. EPC

Chip Freitag is currently an MTS system applications engineer in the embedded processor group at Advanced Micro Devices, where he specializes in networking and telecommunications. Previously, he was a software engineer at Andrew/KMW, where he worked on various 5250 terminal emulation and high-speed page printer emulation products. You may reach him at chip.freitag@amd.com.

Jeff Kirk has spent eight years at AMD as a senior field application engineer specializing in telephone line card applications and embedded processors. Previously, he wrote software for embedded systems, primarily in industrial control and avionics. His software experience covers the gamut from real-time assembler (on most popular micros) to Windows 95 applications written in C++.

SOFTWARE

To retrieve a copy of the Perl script, visit www.io.com/~chipf/perlconvert or the Circuit Cellar Web site.

REFERENCES

AMD, *Fusion E86 CD-ROM*, Publication 19255, 1997.

AMD, *Am186ES/Am188ES User's Manual*, Publication 21096, 1997.

AMD, *Am186/Am188 Family Instruction Set Manual*, Publication 21267, 1997.

B. Brey, *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor*, Prentice-Hall, Englewood Cliffs, NJ, 1997.

W.V. Subbarao, *The 8086/8088 Family Microprocessors: Software, Hardware, and System Applications*, Delmar Publishers, Albany, NY, 1992.

SOURCE

Am186 family
Advanced Micro Devices, Inc.
One AMD Pl.
Sunnyvale, CA 94088-3453
(408) 732-2400
Fax: (408) 732-7216
www.amd.com

IRS

413 Very Useful
414 Moderately Useful
415 Not Useful

Real-Time PC

Ingo Cyliax

Picking a PC RTOS

To implement a real-time system, you have to figure out which RTOS to pick. Using a robot control application as an example, Ingo establishes what fundamental criteria you need to look at first.

Wow! So much has happened this month. I'm starting two new things—a new job and this column. I'm really excited about both. In fact, writing this column is part of my new job description.

And, I get the chance to work more with embedded systems at many levels. At the top end, we develop system-level modeling and verification software. In the middle, we're working on PC/104 modules, which will be used in conjunction with off-the-shelf PC/104 module and software components to build embedded systems. And, we're using some of our tools to develop synthesizable VHDL cores.

I've written for *INK* for a couple years now. You may have noticed my interests range from chip-level designs (e.g., FPGA-based robot controllers) to complete hardware- and software-based systems, like the MC68030 system described in *INK* 86–88.

So, I'm comfortable looking at different levels of detail. I also enjoy the contrast between the true parallelism of hardware objects with the flexibility and complexity of software objects running on a processor.

This meeting of software and hardware essentially describes Real-Time PC. In this column, the embedded PC meets the RTOS. I'll be looking at how real-time embedded PCs solve problems in applications like robot controllers, data acquisition, and more.

Real-Time PC just ran a two-part RTOS 101 series (*INK* 90–91), introducing real-time operating systems, the terminology used, and the typical hardware found in an embedded PC. Those articles will serve as our launching point.

This month, I want to discuss the issues you need to consider in selecting an RTOS for your embedded-PC application. For my application, I'll use a hypothetical robot controller that has some realistic requirements.

Next month, I'll look at the next step—developing the software for the system. OK, let's get on with it.

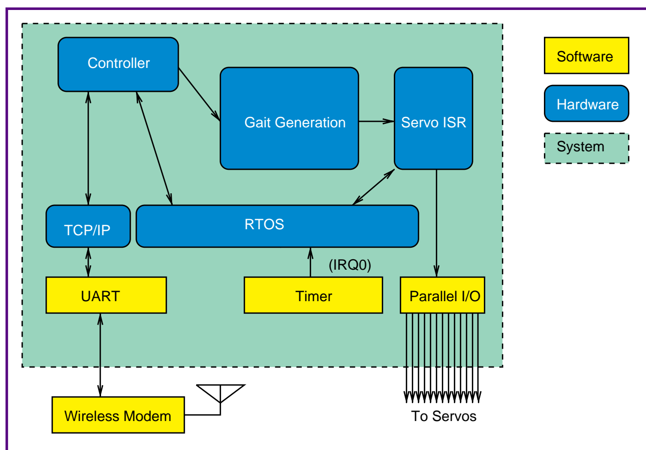


Figure 1—Here's the big picture of our sample system. The controller receives commands from a remote host via a wireless modem and coordinates the robot's leg movements by generating the appropriate timing for the RC-servo actuators.

STARTING LINE

So, where do we start? The beginning always seems like the

best place, doesn't it? For us, that means the specification for the system we're trying to implement.

In other words, we need to identify our system's hardware and software components and know what real-time constraints, if any, it has. We also need an idea of what the target system costs ought to be, and maybe we even need to write some code to model the system's behavior.

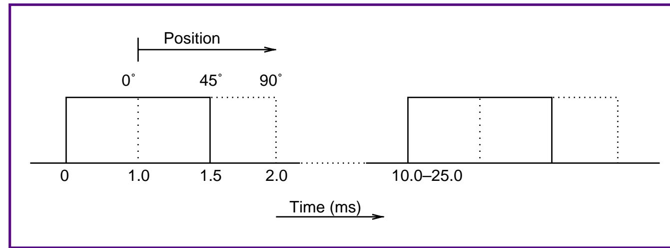


Figure 2—Here we see the timing for an RC-servo actuator used on the robot. The width of the pulse, which needs to be repeated every 10–25 ms, controls the position of the actuator. A 1.0-ms pulse represents 0°, and 2.0 ms represents 90°.

The hardware for this application can be covered by using a standard off-the-shelf PC/AT PC/104 CPU with at least one serial port and 12 parallel I/O ports. As part of the PC/AT architecture, a timer chip can be programmed to post an interrupt to the system. The resolution of the channel 0 timer on a PC/AT is 0.86 µs, which should be sufficient for our purposes.

To make this a little more real, let's spec out a system and try to find an RTOS for it. Since this issue of *INK* is featuring robots, let's consider a robot controller to drive one of the six-legged robots I work with. Figure 1, the system block diagram, shows the software and hardware components.

The system needs to run on a PC/104 platform to be able to fit on the robot and interface with it. It uses 12 parallel output lines to drive 12 radio-control servos, which serve as the actuators. The two actuators for each leg control the up/down and forward/backward functions of the leg.

Besides the actuators, there is also a serial port connected to a radio modem, which sends motion-control commands to the robot from a process on a Unix workstation. These commands are high level, so they focus on things like telling the robot to walk in a certain direction, stop, or walk backwards, and so on.

Our controller has to generate the appropriate leg motion for each of the walking-mode commands. Besides generating the actuation patterns for each walking mode (i.e., gait), it also needs to generate the appropriate control signals for the servos.

RC servos use a pulse-width coded signal, in which the pulse width indicates the desired position of the servo. The servo itself has a position feedback-based controller, which controls its electric motor.

The pulses vary between 1 and 2 ms, encoding positions of 0–90°, and are repeated at 10–20 ms. Figure 2 shows how the position relates to the pulse width. Experimentation has shown that providing four steps, or positions, from 0 to 90° (22.5°/250-µs resolution) is sufficient.

However, these particular servos chatter (i.e., vibrate) when the pulse width signal has too much jitter (typically greater than 100 µs). This means we need to control the jitter in the timing of the servo signal to within ±50 µs.

When the servo is set to 45° (1.5 ms), the signal needs to be between 1.450 and 1.650 ms. Excessive chattering causes unnecessary wear in the servo mechanism and heating.

The wireless modem runs at 9600 bps and provides a communication channel to the serial port of a Unix workstation. Over this channel, I want to run PPP so high-level control processes on the Unix workstation can control the robot.

It does this by establishing a network connection using TCP/IP to the process on the robot that deals with gait control. I don't really care what the software on the Unix workstation does. I'm only concerned with the controller on the robot and how it interfaces.

SOFTWARE

The software for this controller can roughly be split into three components—the controller task, gait generator, and servo drivers.

The controller thread, which starts the ball rolling, initializes the system and networking software and spawns the gait generator. The controller then waits for network connections and handles decoding the high-level commands, which are sent by a process on the Unix host (see Listing 1).

The gait generator coordinates the sequencing of the legs, as shown in Listing 2. Therefore, it needs to communicate with the controller thread and servo driver. The servo driver handles the timing of the servo

Listing 1—The main thread starts the system off and then acts as the communication thread. It receives commands from a remote host via a serial network connection over the wireless modem and signals the gait-generator thread setting the global `Command` ***variable.***

```
int cmdmutex;
int servomutex;
extern int Command;
main(){
  int s,ns;
  struct sockaddr_in sin;
  int slen;
  int GaitThread();
    SpawnThread(GaitThread);
    servomutex = InitMutex();
    cmdmutex = InitMutex();
    s = socket(AF_INET, SOCK_STREAM, 0);     /* establish cmd port
                                                   using TCP/IP */
    sin.sin_addr.s_addr = htonl(MyIPAddress);
    sin.sin_port = htons(MyPort);
    bind(s,&sin,sizeof(sin));
    while(1){                 /* main loop; wait for connection */
      slen = sizeof(sin);
      ns = accept(s,&sin,&slen);
      while(1){                               /* do command loop */
        if((n = ReadLine(ns,buf,sizeof(buf))) < 1)
          break;
        GetMutex(cmdmutex);
        Command = DecodeCommand(buf);
        ReleaseMutex(cmdmutex);}
      GetMutex(cmdmutex); /* remote process has closed connection */
      Command = CMD_STOP;
      ReleaseMutex(cmdmutex);
      close(ns);}}
```

pulses as you see in Listing 3.

Our RTOS needs to respond to timer interrupts with a latency of less than 100 µs. In addition, it should be multitasking, and in particular, it should be multithreaded. It also needs networking support, especially for TCP/IP, as well as a device driver for the serial port to handle PPP.

Obviously, our timing requirements are what make this a real-time project. To address this issue, we need to know what the RTOS's interrupt latency is.

This figure essentially defines how efficiently the RTOS can respond to an interrupt and may include the time the RTOS or application interrupts are blocked during critical sections. The RTOS's interrupt latency is usually given as a time measurement on a given processor architecture.

A proper specification includes the range of interrupt latency (i.e., with or without possible interrupt lock-out times). Typical values are 15–50 µs on a 33-MHz '386. This amount of time should be enough to prevent our servos from destroying themselves. Figure 3 shows a timeline indicating the interrupt latency.

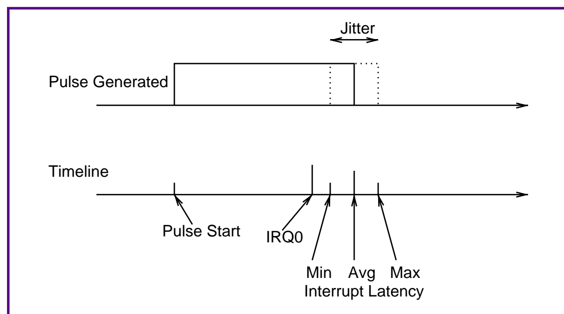


Figure 3—Any nondeterministic timing variation in the interrupt latency of the system introduces jitter in the output signal for the RC-servo actuator. Critical code segments needing to be protected from interrupts usually introduce nondeterminism to the interrupt latency.

But latency isn't the most important issue. When you have several RTOSs, each of which can meet your timing requirement, you have to think about some other factors as well. That is, the fastest RTOS on a particular architecture may not always be the best RTOS for your application.

Although not necessarily important for our application, figuring the throughput of an RTOS is important elsewhere. For example, in multimedia, you may want to make sure you can move the data through the system at the specified rate without starving the DAC for audio or video output. This shows up as obnoxious clicks in the audio and frozen frames on the video.

While calculating this throughput is complex and involves factors like the particular hardware architecture (e.g., bus, peripherals, and processor speed), RTOS vendors claiming multimedia support typically provide some estimate of how well a system built on their architecture might be expected to perform.

There are many timing factors which may affect how a particular RTOS performs in your system. These have to do with how efficient the interprocess communication is implemented as well as rescheduling delays.

Since we didn't put any real-time or throughput constraints on the tasks in the system, it's not an issue here. In a system where tasks may have compute information necessary for real-time response (e.g., an airplane auto-pilot), the system has to continually compute the settings for actuators, while reading sensor and pilot inputs.

Listing 2—The gait-generator thread actuates the legs with stored patterns, depending on the current command mode (e.g., walking forward or stopping). It stores the current position of the actuator based on the pattern in a global data structure.

```
int Command;                                          /* current cmd */
extern SetTime[nCHAN];                     /* time to set servo channel */
extern int MaxSteps[nCMD];                /* number of steps in pattern */
extern int Pattern[nCMD][nSTEP];                       /* gait patterns */
GaitThread(){ /* generate leg actuations depending on current cmd */
  int CurrStep;
  int LastCommand;
  GetMutex(cmdmutex)
  Command = CMD_STOP;
  LastCommand = Command;
  ReleaseMutec(cmdmutex)
  CurrStep = 0;
  while(1){
    Sleep(STEPTIME);              /* check if cmd mode has changed */
    GetMutex(cmdmutex)
    if(LastCommand != Command) CurrStep = 0;
    LastCommand = Command;
    ReleaseMutec(cmdmutex)                      /* set servo channels */
    foreach (i=0;i<nCHAN;i++){
      GetMutex(servomutex)
      SetTime[i] = Pattern[LastCommand][CurrStep];
      ReleaseMutex(servomutex);}
    CurrStep = (CurrStep + 1) % MaxSteos[LastCommand];}}  /* next step */
```

In our system, we also want to use a TCP/IP stack to communicate with a Unix host. This implies that we need to select an RTOS that supports TCP/IP networking.

There are several products out there that offer TCP/IP support. In most RTOSs, this feature is an extra. Some RTOSs even have additional support, such as embedded Web server support.

However, for our robot controller, we only need minimal TCP/IP support. Which brings me to the next issue—memory footprint.

One way to make your system cost-sensitive is to reduce its memory requirements. Recall that I specified a PC/104 CPU for our system. PC/104 boards typically are highly integrated systems, where real estate is used fully.

For example, a PC/AT module which is a potential candidate for my sample controller application is the ZF104Card '386 module from ZF MicroSystems (see Photo 1). This module, designed especially for cost-sensitive applications, only has 2 MB of DRAM, so the memory footprint of our RTOS needs to be pretty small.

RTOSs vary in their memory requirements. Many RTOSs have very small kernels, sometimes as small as 30 KB. On top of this, we have to add modules with the features and facilities we're using in our applications. For example, the TCP/IP stack may take another 100 KB.

Finally, our application takes some static memory for the text and data segments but may also consume memory at run time. Stack space for tasks needs to be computed as well. RTOS vendors can provide you with memory requirements, both static and run-time, for their modules as well as

Listing 3—The servo timing is generated by the servo interrupt service routine, which is called once per tick. It simply generates pulses of programmable duration on the parallel port. Each channel is actuated in sequence, which generates a pulse train of 1.5 ms × 12, or 18 ms, on average. The pulse rate is not critical and varies between 10 and 25 ms, but the pulse width has to be precise to set the positions of the actuators.

```
int SetTime[nCHAN];                              /* value for timer */
int Ticks;                               /* value for current output */
int CurrChan;                                    /* current channel */

Servo_Isr(){        /* only do something when we run out of ticks */
  if(!ticks--){
    outpw(SERVO_PORT, (1<<CurrChan));
    ticks = SetTime[CurrChan++];
    CurrChan %= nCHAN;}}
```

RPC

Photo 1—This PC/104 module, using ZF MicroSystems' PC/AT on a chip, contains everything we need: a 40-MHz '386 CPU, 2 MB of RAM, serial/parallel and disk I/O, BIOS, and a 512-KB SSD in a single module. At less than 2 W, this module is perfect for portable applications, like a robot.

run-time–memory requirements for the task control structures and stack.

But, that's not all. While the devices on our PC/AT-compatible CPU board are standard, this is the exception. We need peripheral boards, which are not standard.

Take the network cards, for example. I'm using the PC/104 module's serial port and a wireless modem as the network interface. Since the serial port is a standard PC/AT 16550 serial port, I can use PPP over it, which many RTOSs have device drivers for.

Now, if I wanted to use a wireless PC Card network adapter with my system by adding a PC Card adapter, I may have trouble finding an RTOS that supports such a card. Granted, wireless network adapters are probably not that much in vogue yet.

But the issue still exists with standard Ethernet cards, for example. If you stick with common Ethernet network adapters, such as an NE2000, you'll have no trouble finding an RTOS that supports this card. So, make sure you allocate resources to write device drivers for peripherals not supported by the RTOS you plan to use.

LICENSING

Another thing to consider is licensing. RTOS vendors usually sell you their toolset and development system, which includes a one run-time license. This package lets you develop a prototype and debug it.

In some cases, like when you are designing a one-off system, this is enough. However, when you're building embedded systems which may be used as a product or installed in several systems, you need to purchase more run-time licenses.

This process varies with each RTOS vendor. Typically, the cost of each run-time license is cheaper the more units you plan to sell, and in some cases, site licenses or unlimited licenses can be purchased.

This pricing can be an important issue in selecting an RTOS. What if the best-suited RTOS for your application has a run-time licensing structure, which makes your product too expensive?

Finally, it seems some RTOS vendors think the weight of their documentation makes it worth more. However, more is not necessarily better. Manuals should be concise and organized in such a way that it's easy to find information quickly. Also, I like a quick tutorial and sample code in any documentation.

I'd also love to see more on-line documentation. Chorus Systems, now part of Sun, publishes their documentation in HTML format so it can be read on any system with a Web browser. This feature makes it easy to look up something quickly, like when I'm sitting in an airplane, frantically trying to finish this article in time for the deadline.

WHAT NOW?

This brings you up to speed about the kinds of issues you need to evaluate when selecting an RTOS for your application. But of course, anytime you need to evaluate anything based on more than one criterion, you need to make tradeoffs.

Some tradeoffs, like performance, are pretty firm. Others, like ideas about documentation, are more flexible.

Another important issue is familiarity. In many cases, assuming the RTOS you've chosen has the performance you need, being familiar with a particular RTOS has a lot of merit.

But, don't let that cloud your judgment. There are several really good RTOSs out there. You can paint yourself into a corner if you don't at least give them a look.

Check out some of the resources listed below. Many are Web and Internet based, so it's easy to obtain and store them on your PC. I usually download information into my notebook to look at when flying.

You're probably wondering which RTOS I ended up choosing for my sample robot controller.

Well, sorry, you'll have to wait. I'll give you some hints next month, when I take a look at software development for RTOSs. RPC.EPC

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

SOURCE

ZF104Card '386
ZF MicroSystems
1052 Elwell Ct.
Palo Alto, CA 94303
(415) 965-3800
Fax: (415) 695-4050
www.zfmicro.com

RESOURCES

Text

J.A. Stankovic and K. Ramamritham, *Tutorial on Hard Real-Time Systems*, IEEE Computer Society reprint series 819, December, 1984.

C. Foster, *Real-Time Programming: Neglected Topics*, Addison-Wesley MicroBooks, Reading, MA, 1981.

C. Vickery, *Real-Time and Systems Programming for PCs*, McGraw-Hill, New York, NY, 1993.

Internet

comp.os.lynx
comp.os.os9
comp.os.qnx
comp.os.vxworks
comp.realtime
www.realtime-info.be
www.cs.umd.edu/~fwmiller/etc/realtime.html
www.realtime-os.com/rtresour.html

REFERENCES

comp.realtime FAQ, www.realtime-info.be/encyc/techno/publi/faq/rtfaq.htm

IRS

416 Very Useful
417 Moderately Useful
418 Not Useful

APC

Applied PCs

Fred Eady

Embedding PC Card

Part 1: The Time Has Come

With PC Cards, the designer gains many additional resources without the cost of incorporating all the functionality in everyone's product and without having to "buy" more real estate. Fred shows you how to make the most of PCMCIA.

You and I use ´em all the time. During the course of a normal business day, I swap Ethernet adapters, modems, Token Ring adapters, and high-speed serial ports in and out of the PC Card sockets of my laptop. To many of us that must code and communicate to pay the bills, the PC Card is simply a tool we cannot do without.

The PC Card of today, known as the PCMCIA card of yesterday, adds flexibility to base PC configurations by enabling you to plug in the function or feature you may need at the time. The obvious advantage is that a single PC Card can be used across many differing embedded and desktop platforms.

Today's embedded-PC environment is quickly moving into the realm of our so-called daily routine. In other words, the embedded product must be capable of taking on different tasks depending on the application it's required to perform at any given moment.

Sure, you could reinvent the wheel by designing and incorporating singular specialized hardware and software for each particular application that must be performed, or you could simply use what's already there—the PC Card. For the embedded application, its time has come.

PC-CARD SKINNY

PCMCIA is a standard that the modern-day PC Card is built on. It's a little confusing because the association chartered to push the PCMCIA is also called PCMCIA (Personal Computer Memory Card International Association).

In the beginning, the physical PC Card was also known as a PCMCIA card, and PCMCIA was the catch-all phrase for the specification, the devices, and the association. I recall how difficult it was back then to remember the acronym PCMCIA because PC Card use wasn't as prevalent as it is now. But since February 1995, PCMCIA cards have been known as PC Cards.

PC Card hardware is divided into categories of cards, sockets, and adapters. PC Cards come in five various types. Table 1 denotes the types, their physical dimensions, and their uses.

As you can see, Type I PC Cards are typically memory cards. The type of memory can vary, but it's usually SRAM or flash memory.

PC Card Type	Dimensions (thickness × width)	Typical Implementation
I	33 × 54.0	Memory cards
II	5.0 × 54.0	I/O cards (network cards, modems)
III	10.5 × 54.0	Hard disks
I Extended	3.3 × 54.0 with up to 40 mm of extended length	
II Extended	5.0 × 54.0 with up to 40 mm of extended length	Pagers, wireless communication adapters (with antennae)

Table 1—Types I and II are probably most familiar, but in my times, I've had the opportunity to use Type II Extended on a daily basis.

The application dictates the type of memory used. Although SRAM is probably cheaper and faster than flash memory, it's power requirements to store and maintain data are high. Designs sometimes give way to the slower but overall less power-dependent flash memory.

Type II PC Cards often need a connector and thus are usually I/O cards. A popular example is today's PC Card modem.

Although Type II PC Cards mimic their desktop counterparts, more often than not, they outperform their traditional card-based cousins. Once again using the common PC Card modem as an example, by placing a more capable UART right on the PC Card modem, you can realize an immediate performance boost if the host computing device is equipped with a less-capable UART.

With the advent of cheaper hard-disk storage, the Type III card isn't as widely used today. A Type III PC Card is typically a miniature rotating head and disk assembly. I recently added a 2.1-GB drive internally to my laptop. So, any Type III hard-disk discussion would be pointless here. Let's move on.

The extended PC Cards perform specialized functions that require them to hang out of the PC Card socket. You've seen these around. They're usually RF or IR based with an antenna or IR emitter/detector attachment. There's even one PC Card modem vendor out there that employs a standard RJ-11 telephone jack here.

That's about all I need to say about PC Cards in a general way. Before we move on to the software elements necessary to implement them, let's talk about a piece of hardware that sits between the PC Card and host system.

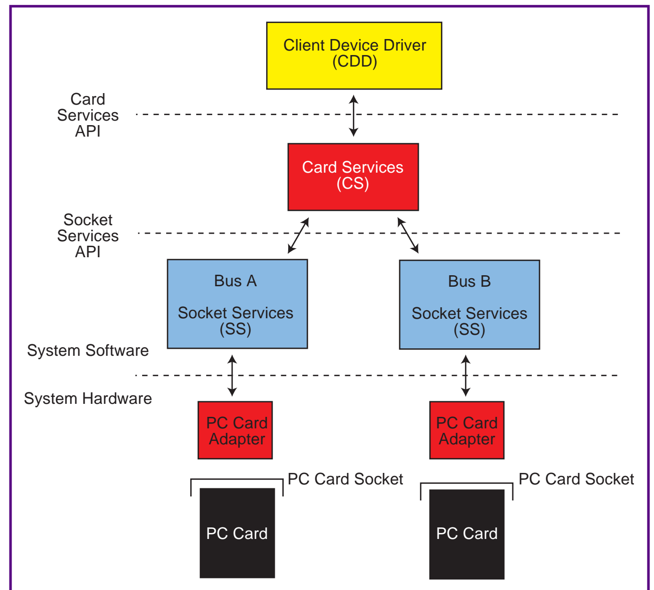


Figure 1—The PC Card adapter is the hardware termination point. Note that you can employ multiple socket services using a single card-services module.

That piece of hardware is an LSI integrated circuit that connects the PC Card socket and host system bus. Called the PC Card adapter, it's nearly always the very first piece of active electronic hardware connected to the PC Card itself. The most commonly used PC Card adapter is the Intel 82365.

PC Card adapters are designed to translate PC Card signals to whatever bus lingo the host system is equipped with. The adapter usually has a standard I/O address of 0x3E0.

The PC Card adapter controls PC Card power, interrupts, and timing functions. Any status change involving a PC Card-generated interrupt (e.g., card detect) is also fielded by the PC Card adapter and routed to the responsible hardware or software. A typical PC Card adapter can handle up to two PC Card sockets.

Just when you think you're in paradise, somebody takes a big bite out of the apple. As you can ascertain from what you've just read and by what you already know about PC Card technology, LSI technology may simplify the hardware piece, but the software side of this could be really hairy.

Essentially, there's this PC Card adapter IC and a socket that must take on a multitude of differing PC Cards, all of which may perform totally unrelated tasks. Again, it's left to the software guys and gals to drive the demons out of the garden.

PC CARD'S SOFT SIDE

In the beginning, each PC Card manufacturer included a piece of software bundled with their PC Card that permitted that card to function in a selected system. This little jewel was (and is) known as a point enabler.

The immediate drawback to point enablers is that they are, by design, machine and PC Card specific. To eliminate the need to depend on each manufacturer to support a piece of specialized software for every computing platform that could use its PC Card, the PCMCIA folks came up with three software modules—card services, socket services and client drivers.

Before the advent of these software services, some manufacturers also included driver

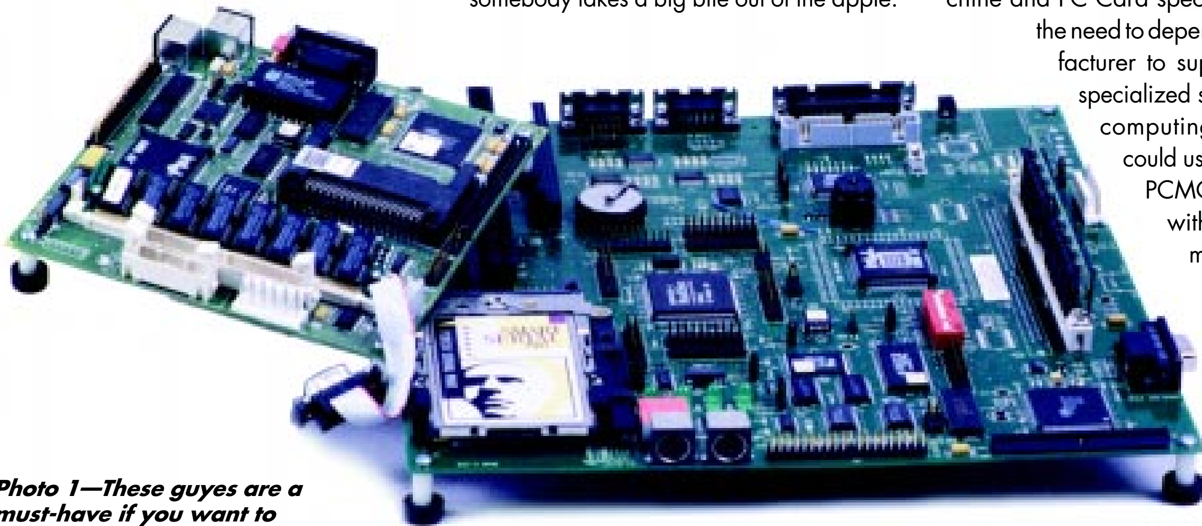


Photo 1—These guyes are a must-have if you want to play PC Card.

routines that set up the PC Card for a particular hardware platform and operating system. One big bugaboo with this was that, depending on the PC Card's function, sometimes you had to shut everything down to use a different PC Card performing another distinct job even if that PC Card was complementary to your currently running application. Bummer.

Card services (CS) is an operating system-specific API that eliminates the need for a hardware-encumbered point enabler. Thus, the dependence on software written for a particular hardware platform is eliminated.

CS usually ships as a component of the operating system or as a device driver that can be easily used in both desktop and embedded environments. It acts as a server to a client device driver (CDD).

Requests for hardware resources are generated by the CDD and routed to the CS. It's up to the CS to keep up with what's available and either grant or deny the CDD request. This client/server relationship means the CDD can be written without any regard to what hardware is on the other side of the PC Card adapter.

PC Card system hardware is managed by socket services (SS). Socket services is tailored to the operational characteristics of the PC Card adapter.

Before SS and CS, the point enabler was responsible for directly interfacing with the PC Card adapter. Therefore, only one point enabler per PC Card adapter could physically exist in a system at a time.

This is also true for SS—sort of. Actually, many socket services routines can be loaded and called on as they're needed. Thus, SS can be located in the system BIOS or included as separate driver routines.

The role of the CDD is dedicated to the PC Card itself. The CDD is responsible for configuring and supporting the PC Card. The CCD relies on CS to coordinate hardware resources and keep the device driver updated with the status of PC Card events. Figure 1 puts these services in perspective.

Once the card is inserted and detected by the PC Card adapter, an interrupt is generated by the adapter. CS, not SS, picks up the request (because SS is not capable of handling interrupts).

Even though SS is most often implemented as a device driver, it's capable of being embedded with the system BIOS, too. That's why it isn't coded for interrupt handling.

CS then calls on SS to root out the cause of the interrupt. In this case, the interrupt was the result of a PC Card insertion event.

SS then informs CS that a card insertion occurred. CS then polls all of the CDDs it knows of to see who wants to service the interrupt.

Depending on the PC Card inserted, one or maybe none of the loaded CDDs step up to the task. Once a CDD accepts and services the interrupt, system resources are allocated to the inserted PC Card.

Conversely, when the PC Card is removed, a similar algorithm is executed that ultimately releases the previously allocated resources in preparation for another insertion.

EMBEDDING THE TECHNOLOGY

Now that you have a good idea about how PC Card hardware and software work together, you're probably wondering how to apply that basic knowledge to an embedded environment.

The first question you may ask is how I enable the PC Card if I don't have a specific operating system running on my

embedded platform.

You may also be wondering about the client drivers. Where do they come from? Being a bit-banger at heart, I asked myself about how I'd go about running and debugging a PC Card application. Well, let's welcome back an old friend to help us out.

Remember the Phar Lap TNT Embedded ToolSuite? Guess what. I just happen to have the latest and greatest Version 9. And guess what. It does PC Card!

It's called the ETS PC Card Support Package. This package contains all the necessary components needed to use PC Card ATA Disks, Ethernet adapters, serial ports, and modems the embedded way.

We've already seen how PC Card technology can enhance even the most mundane of computers. Think about how much value a PC Card socket can add to an embedded solution.

The first thing that comes to my mind is test equipment. With the addition of a PC Card socket and the backing of the ETS PC Card package, you could fabricate a multi-purpose embedded test tool that can change its spots by simply changing its PC Card.

Let's compare the ETS PC Card Support Package's embedded components to the generic software suite set forth by the PCMCIA group.

First of all, the ETS package includes library functions that can be called from applications and supports enablers for most PC Cards in existence today. If a card finds itself alone, there's plenty of example code to steer the PC Card software engineer in the right direction.

But, there is a downside. The ETS Support Package doesn't support sound cards or memory at this time.

On the up side, most of the ETS code can talk to PC Card devices without modification. One example of this occurs when the local file system is an ATA PC Card Disk.

I haven't mentioned it, but most of you know that today's PC Cards support so-called hot swapping—the ability to insert and remove a PC Card without removing power from the host. This feature is especially handy for the embedded application that must change horses midstream without changing saddles.

Well, that's all fine, but ETS doesn't support hot swapping, so don't get real excited about it. The nature of the beast is to have the desired PC Card inserted before and after the target ETS application is run. I'm sure that will get fixed real soon!

For all intents and purposes, the PC Card and PC Card adapter are hardware and will be a constant in our comparison. So, let's start by looking at what I call SS and what ETS dubs as enabler code.

In the case of ATA PC Card Disks, it's possible for the system BIOS to enable the device. In this instance, the application software need not be concerned about the device other than to access its services. The problem is, we all know the majority of embedded systems don't have a built-in BIOS as such, and everything must be handled by the application program.

To handle this dilemma, ETS includes a couple modules that can be linked into the target embedded application. Remember that in the standard PCMCIA spec, CS was defined as operating system dependent but device independent.

Well, it's the same story for the ETS CS module. The other piece of that story is the ETS enabler that parallels the generic SS I told you about earlier.

It's not surprising that the ETS enabler performs the SS functions of providing resource information and allocating IRQ and DMA services to the embedded system. After all, the ETS package is based on the PCMCIA standards. ETS enablers include modules for 3Com and Novell Ethernet adapters, serial lines, modems, and as I mentioned before, ATA disks.

For the embedded system, the enabler code and its functions are important to the operation of the PC Card interface. With that, the ETS PC Card Support Package provides a sample program called CISID (Card Information Structure ID) that lets us determine the necessary information that can be added to an existing enabler to bring the unknown card online. Once the info is gleaned, it's just a matter of recompiling the modified enabler and rebuilding the library.

To make life easier for the PC Card programmer, this process is all automated by batch files included with the ETS package. Listing 1 is an example of the network module rebuild batch file.

On detecting a PC Card, the Realtime ETS Kernel calls each enabler defined in

Listing 1—This is what I've come to expect from Phar Lap—working code complete right down to the compilation parameters.

```
@echo off
REM
REM For sbemb debug symbols add the switch -Z7 to the cl line and
the switch
REM -cvsym to the 386asm line.
REM
386asm -twocase -o smcutil.obj utils.asm
cl -c -I.\inc -MT -Gs -Zl -W3 -Og -Oi -GF smc16.c
lib -out:eth-smc.lib smc16.obj smcutil.obj
386asm -twocase -define _PET9_ -o 3comutil.obj utils.asm
cl -c -I.\inc -MT -Gs -Zl -W3 -Og -Oi -GF 3c509.c
lib -out:eth-3com.lib 3c509.obj 3comutil.obj
386asm -twocase -define _NE2K_ -o ne2kutil.obj utils.asm
cl -c -I.\inc -MT -Gs -Zl -W3 -Og -Oi -GF ne2k.c
lib -out:eth-ne2k.lib ne2k.obj ne2kutil.obj
386asm -twocase -o smc9util.obj utils.asm
cl -c -I.\inc -MT -Gs -Zl -W3 -Og -Oi -GF smc9.c
lib -out:eth-smc9.lib smc9.obj smc9util.obj
cl -c -I.\inc -MT -Gs -Zl -W3 -Og -Oi -GF ser16550.c
lib -out:ppp16550.lib ser16550.obj
```

Listing 2—Here's a typical ETS enabler record. If the PC Card is to be found, this record must exist.

```
static isa_db_t isa[] = {
  { CS_ISA_VERSION, "3Com 3C589 LAN Adapter", {
    "3Com Corporation", "3C589" } },
  { CS_ISA_VERSION, "3Com 3C589D LAN Adapter", {
    "3Com Corporation", "3C589D" } }, };
```

Listing 3—To use the ID database for EtsCSIsA, insert the card in question (and only that card) into a socket. Run this program, redirecting its output to a file select the single line best fitting your situation and delete the rest. Insert the useful line into the ID database in your enabler.

```
static isa_db_t isa[] = {
  // Socket 0;
  { CS_ISA_FUNCTION, "Network Adapter", {
    (void*)CISTPL_FUNCID_NETWORK } },
  // Change fields to NULL if they seem too specific
  // The more generic fields come first, the more specific come
last
  { CS_ISA_VERSION, "Put your card's name here", {
    "PMX   ", "PE-200", "ETHERNET", "R01", } }, };
```

the system to determine the correct enabler for the requesting card. The enabler first queries the PC Card for the CIS data and then compares what it receives to its own internal database.

This CIS data resides within the PC Card proper and follows a predetermined format, which is set forth by the PC Card standard. The local database used by ETS is the `isa_db_t` data structure found in the include file `EMBCS.H`.

Listing 2 is a snippet of the enabler record for the 3Com card. All of the network card enabler entries are similar in content. CISID is coded to query any PC Cards on the embedded target and create an `isa_db_t` record for the card.

Listing 3 is a sample of the output produced by CISID when run against an unrecognized Ethernet PC Card. Once the PC Card is properly recognized and configured, it's business as usual on the I/O ranch.

ALL THAT HARDWARE

The TNT Embedded ToolSuite V.9 also brings another old friend to the PC Card party—Intel's EXPLR2. Recall that the EXPLR2 is almost a perfect embedded PC/AT platform. Just about everything on this embedded eval board works just like the desktop.

The latest ToolSuite provides native support for the EXPLR2 and its onboard PC Card socket and adapter. That is, there's already a kernel assembled and just waiting to be loaded. You've already read about the extensive PC Card support.

In addition, the new TNT ToolSuite can support the EXPLR1 and its PC Card socket, too. Photo 1 is a shot of the twins.

The Phar Lap folks didn't forget about Bill, either. This version has the capability of running from within the Microsoft Developer Studio using Microsoft C++ V.5. Another blow to the DOS command line.

NEXT TIME, TEST TIME!

Although the TNT PC Card Support Package isn't perfect, it's well thought out. Phar Lap tends to have a logical approach to solving embedded problems with their products. The new support included within V.9 is just another example of their good code and good coding tools.

I'm about out of paper, so here's what we're gonna do next time. Recall that I mentioned something about test equipment. Well, you know what that means. In the next installment, I'll turn one of the twins into a dedicated test instrument via its PC Card interface. 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

TNT Embedded ToolSuite V.9
Phar Lap Software
60 Aberdeen Ave.
Cambridge, MA 02138
(617) 661-1510
Fax: (617) 876-2972
www.pharlap.com

EXPLR2, EXPLR1
Intel Corp.
5000 West Chandler Blvd.
Chandler, AZ 85226-3699
(602) 554-8080
Fax: (602) 554-7436
www.intel.com

Microsoft C++ V.5
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
(206) 882-8080
Fax: (206) 936-7329
www.microsoft.com

IRS

419 Very Useful
420 Moderately Useful
421 Not Useful

FEATURE ARTICLE

Gordon Dick

Motor Speed Control with a Microtwist

If you need to modify an analog speed-control circuit, you're in for a major redesign. But if your system is digital like Gordon's, adding an extra feature isn't so complicated. Hop onboard to learn how to develop speed control with a 68HC11.

Some years ago, an analog speed-control circuit was designed to control the speed of a line-powered universal motor driving a woodworking machine. The circuit worked fine on the bench, but when it was connected to the machine, it was painfully apparent there should have been a provision to control the rising rate of the setpoint.

The control system was so powerful that if the setpoint changed too rapidly, the motor caused the drive belt to slip and squeal loudly. However, that sort of a change requires a redesign of the PCB.

The system has never been changed. Had it been digital, like the one I talk about in this article, adding an extra feature would have been simple.

REVIEWING PHASE CONTROL

It's often necessary to control how much power is delivered to a piece of line-powered equipment. The most common example is light dimmers.

The dimmer adjusts an incandescent lamp's brightness by controlling where in the half cycle of the line voltage a switch (e.g., a triac or SCR) turns on. The amount of the half-cycle angle is called the conduction angle. The waveforms in Figure 1 illustrate different conduction angles and lamp intensities.

Varying the voltage to a load by changing the conduction angle is a technique also used to control the speed of universal motors. To do this, you need a unit, available in many hardware stores, that's basically a light dimmer with more heatsinking, a wall plug-in cord, and a tool plug-in receptacle.

With it, you can control the speed of a drill, router, or any tool powered with a universal motor. However, this method of speed control doesn't include any feedback. As the load on the tool increases, the tool slows down because the voltage to the motor is constant.

SYSTEM OVERVIEW

Figure 2 illustrates a closed-loop speed-control system where most of the work is done by an embedded controller (an 'HC11, in this case). Only a little additional circuitry is required.

An analog setpoint is produced at the wiper of the potentiometer and immediately converted to a digital quantity. A setpoint can be provided in various ways, but a control knob suits me best.

The 'HC11's ADCs are eight-bit units. That's often a limitation, but not here. It simply means there are only 256 different setpoints, but the control-loop calculations are done in 16 bits.

To build a feedback system, you first need to measure the quantity you're trying to control. Motor speed-control systems often use a tachometer generator to provide an output voltage proportional to speed.

I use an optical interrupter to produce a pulse train with a frequency proportional to motor speed. And, I have some good reasons for this.

For one, an optical interrupter is cheaper than a tachometer, especially if you build it. As well, a tachometer needs another ADC, so the 'HC11's eight-bit ADCs is a disadvantage because this one is inside the control loop. And finally, fitting an optical interrupter into a system is mechanically easier than coupling a tachometer to it.

In Figure 2, the 'HC11 is fed a signal from the optical interrupter, whose frequency varies directly with motor speed. The 'HC11 includes a timer that can easily measure this signal's period and that can be used as a feedback quantity.

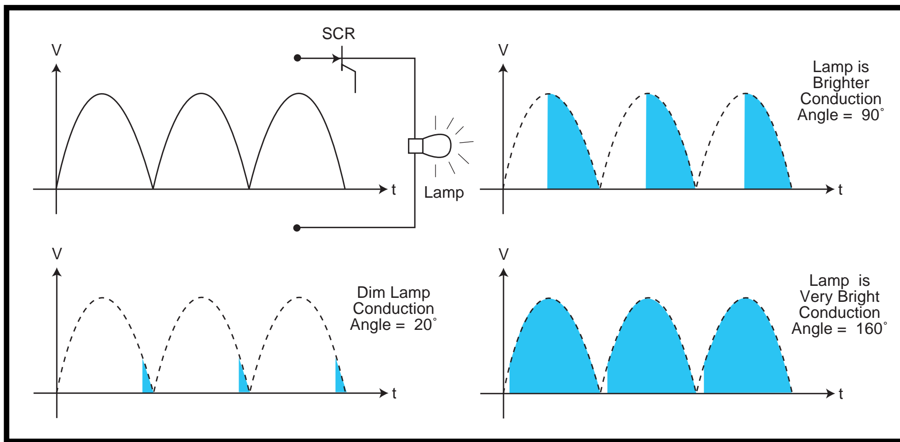


Figure 1—*Varying the conduction angle changes the lamp's brightness.*

Once a feedback signal (or number) is available, it can be compared to the setpoint signal. A simple subtraction produces an error signal, which is then fed to a control algorithm.

The calculations executed in the control algorithm determine the kind of control implemented. In other words, they make the controller a P, PI, PD, or PID, where P is proportional, I is integral, and D is derivative.

A simple proportional control algorithm multiplies the error signal by a constant. The signal from the control algorithm then provides a signal that is appropriately delayed relative to the power-line zero crossings, so the conduction angle of the voltage applied to the motor causes it to be driven in a way that minimizes the loop error.

For example, if the load on the motor increases, causing it to slow down, the control loop measures the lower motor speed and delivers a drive signal to the motor driver. This driver then feeds more voltage to the motor to make up for the lost speed.

TRANSLATING INTO CODE

Since the code for this project is too long to include here (see Software for download information), I want to explain how the block diagram in Figure 2 might be implemented.

I built the motor drive signal unit first. Since this unit must produce signals delayed a specific amount relative to the line-voltage zero crossings, it makes sense for the unit to be interrupt driven. So, the unit runs at every line-voltage zero crossing and produces a pulse after a set delay passed to it.

Suppose the delay number passed to the motor drive signal unit is 4 ms. Then, 4 ms after the line-voltage zero crossing, the motor driver receives a signal to turn on the motor and apply power to it for the remaining half-cycle.

Decreasing the delay number increases the motor voltage by applying power to the motor earlier in the half-cycle. The motor drive signal unit runs as an interrupt service routine (ISR), and the remaining code to implement Figure 2 runs when it finishes.

The setpoint unit gives instructions to the ADC system to convert the setpoint signal continuously. This task is handled by a dedicated system, so it doesn't take any processing time away from the real-time loop.

So, you get a value for the newest setpoint simply by reading a register. (An average setpoint is obtained from four ADC readings.) Shifting the setpoint reading left by eight bits forces it into a 16-bit value.

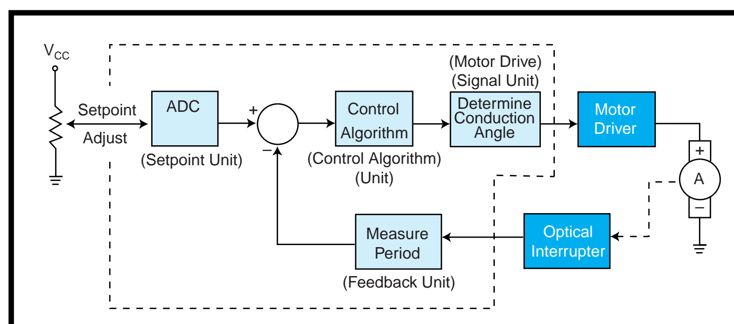


Figure 2—*The 'HC11 can do nearly all the speed-control tasks. It performs the functions in the dashed box.*

To measure the period in the feedback unit, the timer subsystem is configured to set a flag on every leading edge of the optical interrupter's signal. An internal clock is read each time a flag is set, and the difference in internal clock readings is proportional to the signal period. This number is passed to the control algorithm.

Error detection is done in the control algorithm (i.e., the 16-bit setpoint has the 16-bit period subtracted from it). The error signal passes to the control algorithm, where it is either multiplied by a constant which calculates a new delay number for the motor drive signal unit if the motor is running too slow, or the delay is increased to maximum if the motor is running too fast.

At maximum delay, the motor does not get any voltage applied to it for a half cycle of the line. The delay number calculated is used by the motor drive signal unit the next time the ISR runs.

GETTING DOWN TO IT

Now, I just need to make the concept a reality. That shouldn't be hard to do. The 'HC11 is going to do most of the work! Well, sort of.

There's a fair amount of code to create, and this is after all a real-time control loop. Since I'm short on the tools standard in the real-time embedded-control business (and since my code never works the first time), I approached things systematically. I modularized the code building and testing into small manageable units.

During the course of this project, I created many small test files for various functional blocks. I'll discuss four major functional blocks of code.

The 'HC11 makes the setpoint unit easy to deal with. The A/D subsystem is initialized to convert a single analog input continuously. When the real-time loop needs a setpoint for the error calculation, it's available in a register.

As for the feedback unit, measuring a signal's period is handled nicely via the 'HC11's timer subsystem. When a period measurement is

required, a free-running clock is read at an edge of the signal to be measured and read again at the next similar edge.

The difference between free-running counter values is directly related to period. (The code documentation for this unit has a more detailed description of period measuring.)

The control-algorithm unit appears simple at first. Do a subtraction and multiply by a constant. Ah, but there are more details to deal with.

Because the 'HC11 `mul` instruction isn't signed, negative and positive errors must be handled separately. And since the phase-control method of driving the motor can't force the motor to slow down (friction or the external load does that), the control actions are different for positive and negative errors.

When the control loop senses the motor is going too slow, the voltage to it is increased. But when the motor is going too fast, the motor voltage reduces to zero so it can slow down to the desired speed as quickly as possible.

In this unit, I need to make decisions regarding the proportional gain (i.e., the proportional band). In other words, over what range of errors will proportional control exist, and beyond what bounds will I force the equivalent of analog saturation?

Fortunately, as long as the code to take these issues into account is included, the proportional band can be tuned when the system is finally operational.

The 'HC11's output-compare (OC) subsystems make producing the motor drive signal rather straightforward. This code runs as an ISR when the *IRQ input is driven low by a synchronizing signal at each line-voltage zero crossing.

The OC subsystem uses a free-running counter. When the counter number matches the number passed to the OC subsystem, a flag is set. The 'HC11 has five OC subsystems, and the ISR producing the motor drive signal uses OC2.

At each line-voltage zero crossing, the OC2 output is forced low to prepare a signal for the motor drive circuit to apply power to the motor. A set time to wait before applying power to the motor is passed to the ISR, which is used in OC2. So when the ISR runs, after the delay calculated by the control-algorithm unit, a drive signal is fed to the motor drive circuit via OC2.

GET OUT THE SOLDERING IRON

Until now, all the code has been tested on the 'HC11 development board at my desk. All the signals appear to behave correctly on the oscilloscope.

But, there's no motor whirring and responding to load changes and setpoint changes. It's time to build a prototype (see Photo 1).

I didn't anticipate many difficulties because I've built analog speed-control units before. I just use the motor drive circuit and optical interrupter circuit from before, and substitute the 'HC11 for the rest. Simple, right? Not quite.

Since this prototype will eventually run on its own using code stored in the internal EEPROM of the 'HC11, the prototype is wired so the signals produced by the 'HC11 can be supplied from the development board via an interconnect cable. Of course, there is no 'HC11 in the socket on the prototype board when the system is operated this way.

Only after everything is working satisfactorily in this mode will I re-

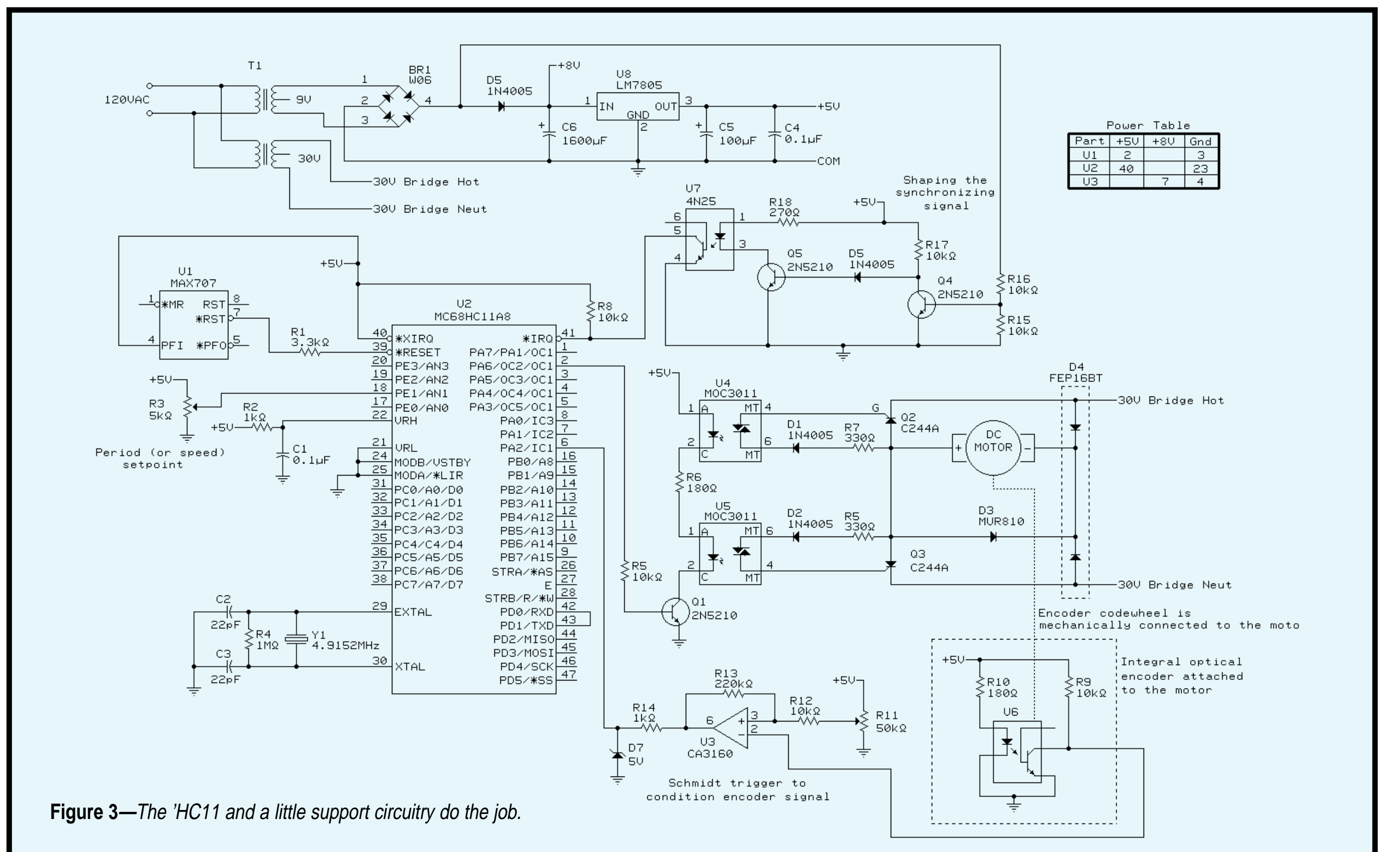


Figure 3—*The 'HC11 and a little support circuitry do the job.*

move the umbilical cable from the development board and install an EEPROM-programmed 'HC11 in the prototype board. This tack turned out to be a big time saver.

For a while, the code worked fine running out of RAM on the development board but not out of EEPROM. (This was when I was learning about relocatable code. More on that later.)

There could have been a lot of frustrated chip swapping at this point with bent pins and blue air. Happily, at least the bent pins were avoided.

Figure 3 shows the schematic for the prototype. Eventually, this circuit will be used to control universal motors in the 0.5–2-hp range. However, working with a smaller motor initially makes more sense.

Since I didn't have a small universal motor, I opted for a small DC motor to be driven from a phase-controlled bridge. This makes the motor drive circuit a bit more complicated, but I had a smaller motor on hand.

As a bonus, my DC motor also has an optical encoder integrally mounted. Therefore, I didn't have to build an optical interrupter circuit and code wheel, which I thought would save some time. But, no! It turned out that this optical encoder didn't produce TTL-compatible signals.

To make things worse, the small signal it produced was superimposed on 4.5 V, making it difficult to extract. I'm sure there are op-amps with input common mode ranges that include the positive supply, but I didn't have any, hence the somewhat unorthodox way of powering the Schmitt trigger.

So, the motor with the integral encoder didn't save much time. It just

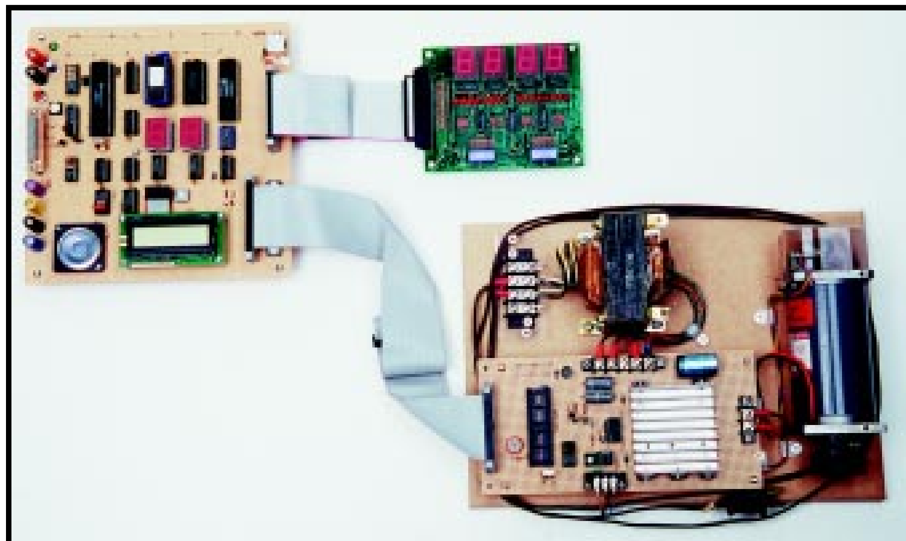


Photo 1—*The development board connects to the prototype via the umbilical cable. Note the 'HC11 is missing in the prototype board. During debugging, all of the displays were used to show the values of the variables.*

changed the tasks. Ordinarily, the Schmitt trigger and its associated components aren't required.

The optocoupler U7 requires some explanation also. As I mentioned, a lot of initial testing was done with no 'HC11 in the prototype board and a cable to the development board, which provided the signals.

At first, I got a lot of spurious resets on the development board when the motor was running. I could never catch a noise spike, but I figured the noise had to be getting into the development board via the *IRQ signal lines.

Optoisolating the *IRQ signal and ground lines between the prototype and development boards solved the problem. Since I may yet tweak the code for this project, I decided to leave the optocoupler in. However, it shouldn't be necessary on a production version.

One final comment about Figure 3. Since usually the motor would be powered directly from the 120-V line, the winding on T1, which produces 30 V for the motor bridge, wouldn't be needed.

CLOSING THE LOOP

Experience has taught me that even though I've tested the code and have all the electronics working, the system will not work. Some part of the code isn't quite right, and I don't know that because the feedback loop is not yet closed.

Systems that incorporate feedback are particularly difficult to troubleshoot with the loop closed. So, the trick is to fake a closed-loop system.

If you have a signal generator, use it to simulate the optical interrupter signal. In other words, don't connect the signal from the motor encoder, but rather use a signal generator instead.

I didn't have a signal generator, but I did have another motor with an encoder attached—a mate to the motor/encoder.

By powering this motor with some fixed voltage, I produced a pulse train just like the signal generator. When the loop behaves correctly, the motor connected to the phase-controlled bridge runs at a speed dictated by the setpoint and the faked encoder signal.

If the faked encoder signal's frequency is reduced, then a load is simulated on the motor, slowing it down. The feedback loop should respond by increasing the motor voltage. If this control action is not happening under these conditions, you can be sure the actual system won't work either.

Eventually, I got the problems corrected, and the faked feedback loop behaved appropriately.

Getting this system to work was a bit of a high. Without the proper tools for this kind of development, troubleshooting was slow.

CUTTING THE CORD

Up to this point, I've done everything using the 'HC11 in the development system shown in Photo 1 with a cable connecting it to the prototype board. And, the code being executed was contained in the development board's RAM.

Since I got the feedback system working correctly, it should be simple to transfer the code into EEPROM and have a stand-alone system. Oops, wrong again.

Getting the code into the EEPROM went quite smoothly. Motorola's reference manual was helpful [1], providing clear examples.

To load code, I built a program that copies part of itself somewhere else (i.e., into the EEPROM). Then, I made the part to be copied the motor speed-control code.

Because the motor speed-control code was assembled and linked as part of a larger program and then copied into EEPROM, it is essential that it be relocatable. To run code in EEPROM, the 'HC11 has to be started in Special Bootstrap Mode. This meant adding jumpers to the development board.

Also, the interrupt vectoring is different, so my code had to be changed. So, put the jumpers in the bootstrap position, hit Reset, and it should control speed, right?

Wrong. Troubleshooting just got an order of magnitude more difficult. At least before, I could put a breakpoint in. Now, I can't even do that.

By embedding some diagnostics, it appeared the code executed once and didn't `jmp` back to the top the way it was supposed to. After puzzling over this a while, I remembered hearing that the code transferred to EEPROM

#129

had to be relocatable. Suddenly, it all made sense.

That `jmp` instruction jumped to an area where there is no code, interpreting what it found as opcode and ending up in the ditch. "Relocatable" took on a clearer meaning! So, I changed `jmp` to a `bra`, and it worked.

Now, it's time to cut the cord. Take the 'HC11 from the development board, install it in the prototype, and disconnect the ribbon cable. Yes, it works!

ADDING FEATURES

By now, I'd created what seemed like quite a bit of code. And, I hadn't even spent much time shaving bytes here and there. To my surprise, the 'HC11's 512-byte EEPROM was only a little over half full.

The primary feature I want to add is the one that got left out of the analog design I mentioned at the start. In other words, I don't want the system to respond instantly to an abrupt change in setpoint.

To control how the system responds to a setpoint change, I added code that keeps the setpoint in the control algorithm until the real-time loop has been executed a set number of times.

Then, the setpoint in the control algorithm is only increased or decreased by one. This continues until the external setpoint matches the one used by the control algorithm. Rapid changes in the analog setpoint now produce a very sedate response from the prototype board.

There's another feature I worked on but chose not to include here. The best way to describe it is to call it a watchdog.

At very low speeds, the ISR won't complete a period measurement before the next *IRQ appears. When this happens, the *IRQ is masked.

This leaves the OC2 output high too long causing a full half cycle of line voltage to be applied to the motor before the following *IRQ is honored. As a result, there is a speed below which the motor runs rough.

I wrote code that monitors the time remaining from the last *IRQ, and if the ISR doesn't complete in ~8 ms, the watchdog aborts it. When this happens, no new values are calculated and stale numbers are used in the control algorithm. This keeps the motor from running rough at low speeds, effectively setting a low-speed limit.

NEXT STEPS

This project was a valuable learning experience. As I mentioned, the term "relocatable" has taken on a graphic meaning for me now.

I also gained a new respect for the 'HC11. Being able to use a single-chip micro as a design solution is just plain cool.

Hopefully, my next foray into a single-micro solution to a control task won't take so long since I can build on what I learned here. And, I keep telling myself that once it's working, it'll be so easy to add a feature. ■

Gordon Dick is an instructor in Electronics Technology at the Northern Alberta Institute of Technology in Edmonton, AB, Canada. He is a member of the American Institute of Motion Engineers and is the first Canadian to obtain the Certified Motion Control Specialist (CMCS) designation. You may reach Gordon at gordond@nait.ab.ca.

SOFTWARE

Complete source code for this article is available via the Circuit Cellar Web site.

REFERENCE

Motorola, *M68HC11 Reference Manual*, Rev. 3, 1991.

SOURCE

68HC11
Motorola
MCU Information Line
P.O. Box 13026
Austin, TX 78711-3026
(512) 328-2268
Fax: (512) 891-4465
freeware.aus.sps.mot.com

I R S

422 Very Useful
423 Moderately Useful
424 Not Useful

#131

DEPARTMENTS

66 MicroSeries

74 From the Bench

80 Silicon Update

EMI Gone Technical

MICRO SERIES

Joe DiBartolomeo

Suppression Components

Part 2 of 4

Sometimes it feels like a case of eenie-meenie-miney-moe when it comes to figuring out which suppression component is best for you. Time to chat with Joe. He has some guidelines that help you find the best fit.

I started this MicroSeries with a look at the most common sources of transient EMI and their representative waveforms. This month and next, I want to examine the components that protect electronics from these transients.

There are many ways to protect circuits and systems from transients. Protection techniques range from simple series resistors to the specialized and specific, like the quarter-wave shorting stubs used in telecom applications, and everyone has their own favorite methods and components.

Of course, due to the varied nature of EMI transients, no single technique or component can protect equipment from all transients. But, you'd be surprised how many designers don't want to modify their EMI-transient protection, even when they discover their equipment is underprotected.

Recently, I talked with a designer who had been using zener diodes on I/O lines for years. When I pointed out that TVS diodes would be a better solution, he agreed to think about it.

After considering the TVS diodes versus the zeners (a topic I'll cover next month), he came to the same conclusion I did—his equipment was underprotected. But, he still stuck with the zeners. Why? Because they had worked for so many years without a problem.

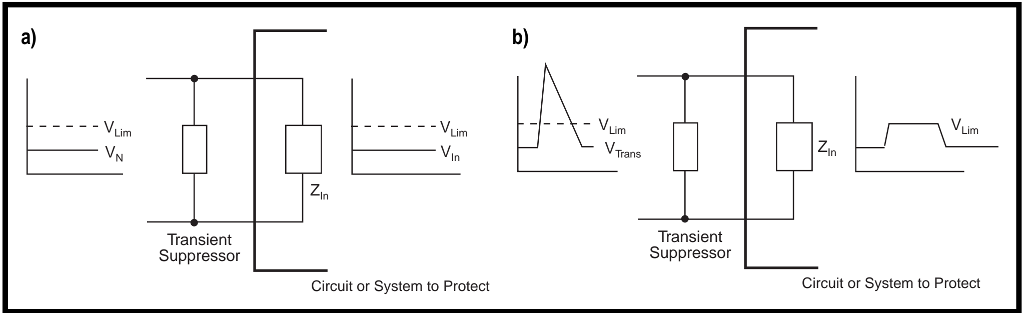


Figure 1a—*Transient-suppression devices are normally connected in shunt across the circuit or system being protected. Under normal operating (nontransient) conditions, the transient suppressor has no effect on the system.* ***b—****When a transient voltage* (V_{Trans})*, appears, the transient-suppression device limits the voltage to* V_{Lim}*, ensuring the input voltage does not exceed* V_{Lim}*, thereby protecting the circuit or system.*

Now, his basic argument—if it ain't broke, don't fix it—is a design philosophy I strongly adhere to. But, zeners are much too slow for his application, and TVS diodes are definitely the better choice.

What's going on? Why is his equipment still working?

One possibility is that the worst-case transients he designed for didn't appear. But, this option doesn't seem too likely after several years.

The more reasonable explanation is that other circuit conditions are mitigating the transients' effects, thus enabling the zeners to provide adequate protection. Recall that the effect of the transient depends as much on the transient's receptor as it does its source and coupling path. Either way, it's a problem waiting to happen.

For example, one circuit condition possibly enabling the zeners to provide proper protection is IC technology. A zener may adequately protect older, robust 5-V TTL chips, but it may not achieve the same level of protection for today's newer, smaller, faster, lower voltage chip technology.

As you know, selecting components to protect equipment from EMI transients requires some degree of knowledge and understanding. But in fact, if care is taken in the design stages, selecting the appropriate component is often fairly easy.

Although there are many more design issues to be covered and I'll return to the topic of design philosophy, now is an appropriate time to introduce the components most commonly used to protect against transient EMI.

COMMON COMPONENTS

The two most commonly used transient-suppressor types are arc discharge and semiconductor. The arc-discharge group includes air spark gaps and gas discharge tubes. Semiconductor transient-suppression devices include diodes, zener diodes, transient voltage suppression (TVS) diodes, thyristor/zener combinations, and metal oxide varistors (although MOVs aren't actually semiconductors).

The devices you employ to protect your circuits from transients depend on the characteristics of the device itself, the nature of the transient, and your circuit or equipment parameters.

The best protection technique is to divert the transient energy away from the component or circuit and into ground. Therefore, transient-suppression devices are normally connected in shunt across the component or circuit being protected, as shown in Figure 1. Transient-suppression components are triggered by the voltage across them, another reason for them being employed in a shunt arrangement.

Under nontransient operating conditions, the suppressor component doesn't affect the operation of the protected circuit, behaving (ideally) as an infinite impedance. When a transient appears across the suppression component, it limits the voltage across and diverts the energy away from the circuit.

Transient-suppression devices fall into two broad categories—clamping devices that absorb transient energy and crowbar devices that reflect it.

CLAMPING DEVICES

As the name implies, clamping devices simply clamp the voltage to a maximum level. Figure 2 shows the voltage/current (VI) curve of a typical clamping device (e.g., a MOV or silicon avalanche diode).

Ideally, the device looks like an open circuit until the clamp voltage is reached. At that point, the voltage is clamped and the device can handle infinite current. The VI curve of practical clamping devices can be divided into three regions—high impedance, transition, and clamping.

When the voltage across the devices is less than V_{Nom}, the device looks like a high impedance drawing minimal current, I_{Nom}. When the voltage across the device begins to increase towards V_{Clamp}, there is a transition region between V_{Nom} and V_{Clamp}.

The breakover voltage is the point at which only a small voltage increase is required to greatly up the device's current-handling capabilities. Once the voltage across the device is greater than the breakover voltage, we're into the clamping region and the device clamps the voltage across it to V_{Clamp}.

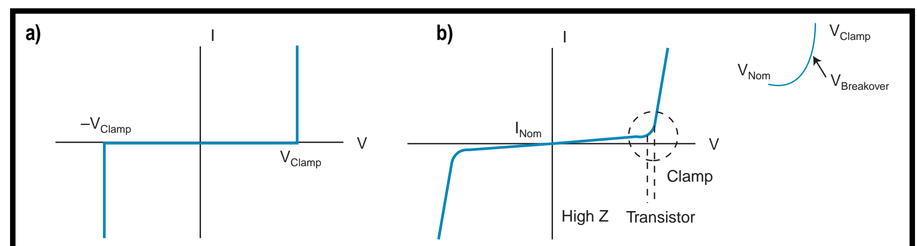


Figure 2a—*Note that the ideal device draws no current until* V_{Clamp} *is reached, at which point the device turns on instantly and can handle infinite current.* ***b—****The VI curve for an actual clamp suppressor shows that in the off state, it draws nominal current,* I_{Nom}*, has a transition to clamping, and a limit to the current it can handle.*

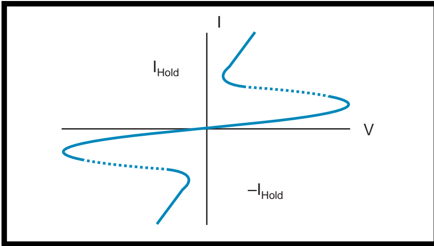


Figure 3—*Here's a typical VI curve for a bidirectional semiconductor crowbar-type surge-suppressor device. Crowbar devices reflect much of the transient energy back to the transient source, allowing them to dissipate much more energy than clamping devices.*

The regions of the VI curve enable us to compare the characteristics of clamping devices. The high-impedance region lets us compare the characteristics in the off state (e.g., MOVs have a higher I_{Nom} than zener diodes).

The size of the transition region and the *di/dv* of the VI curve in the transition region indicates the speed of the device, indicating the types of transients it protects against.

For example, the transition region for a MOV is relatively large and its *di/dv* is relatively slow, indicating a soft transition to clamping. The TVS diode's transition region is very small and the *di/dv* very large, permitting the TVS diode to transition hard into the clamping region.

The clamping region tells how much energy the device can handle. Diodes dissipate the transients in their junctions, which account for only a portion of their mass. Therefore, they can turn on quickly and hard.

Clamping devices that use all of their mass to dissipate transients (e.g., MOVs) can handle much more energy per unit device size than junction-dissipating devices can. Of course, the larger areas make MOVs much slower.

CROWBAR DEVICES

The VI curve for a typical bidirectional semiconductor crowbar device is given in Figure 3. Under normal operating conditions, the crowbar device is similar to the clamping device. It presents a high impedance and draws a nominal current, I_{Nom}.

When the voltage across the crowbar device increases to $V_{Breakover}$, the impedance of the device begins to drop and conduct energy away from the device being protected. As the voltage across the devices increases, it reaches a foldback voltage, at which point the voltage across the crowbar device drops significantly.

The crowbar device then begins to reflect much of the transient energy back to the transient source. Once the transient is removed, the crowbar device returns to its high-impedance state only after the current through it drops below its holding current, I_{Hold}.

As with the clamping devices, the VI curves of the crowbar devices provide insight into the nature of the device as well as which device is best suited for a particular transient threat.

Crowbar devices inherently handle greater amounts of energy than clamping devices. Crowbar devices reflect a great deal of transient energy back to the source, whereas clamping devices dissipate the transient energy by junction heating.

The clamping ability is $V_{Peak} \times I_{Peak}$ per unit time. However, crowbar devices have the disadvantage of their hold currents.

AIR-GAP DISCHARGE DEVICES

Air-gap discharge devices are generally used between the line to be protected and earth ground. The basic construction is two electrode tips made of metal or carbon separated by an air gap.

In normal operating conditions, these devices have virtually infinite impedance, drawing no leakage current and therefore not affecting the circuit being protected. These devices don't begin to conduct until their breakover voltage is reached, at which point they arc across, diverting the transient energy to ground and protecting the line.

The breakover voltage depends on the type of electrode and size of the air gap. Environmental conditions (e.g., humidity) also affect the breakover voltage.

The breakover voltage can be as low as 100 V or as high as thousands of volts. However, the uncertainty of the breakover voltage means you need to take some care in designing with these devices.

As an example, let's assume the working voltage on an I/O line is 75 V and an air-gap arc device with a breakover voltage of 100 V is protecting the line. The first thing you notice is that unless your I/O line can handle a transient of 100 V for 1 µs (typical turn-on time), you need secondary protection.

But, you must also be aware of the environmental conditions under which the breakdown voltage is specified. Otherwise, you'll have more or less protection than you think. Also, the *dv/dt* of the transient affects the breakover voltage (more about this when I discuss gas-tube discharge devices).

The characteristics and low cost of air-gap devices make them ideal for primary protection against lighting transients, in which the transient suppressors are replaced as part of scheduled maintenance, and or in applications where transient suppressors are replaced after every thunderstorm. But as I mentioned, you normally need secondary protection.

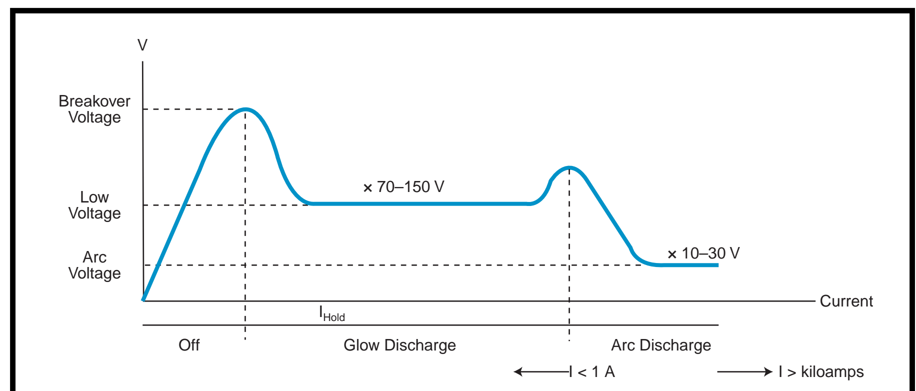


Figure 4—*Here, I used an IV curve rather than a VI curve, since it's easier to see the behavior of the gas tube. However, the crowbar action is harder to see. Note that the current and voltage axes are not to scale.*

GAS DISCHARGE TUBES

Gas discharge tubes, also known as gas-tube arresters, are similar to air-gap suppressors but are designed to overcome several of their disadvantages.

These devices consist of two or three electrodes enclosed in a ceramic tube, filled with inert gas and hermetically sealed. This construction eliminates the uncertainty caused by environmental conditions and enables the breakdown voltage to be easily controlled.

The breakdown voltage is a function of the gap distance between the electrodes (on the order of 1 mm), the gas in the tube (normally a mixture of argon and hydrogen), and the gas pressure (normally 0.1 bar). Devices are available with breakover voltages ranging from 80 to several thousand volts with current ratings in the kiloampere range.

The gas tube is normally connected in parallel with the line to be protected—one end connected to the line being protected and the other end to ground. As long as the voltage across the device is below the gas tube's breakover voltage, the gas tube has virtually infinite impedance.

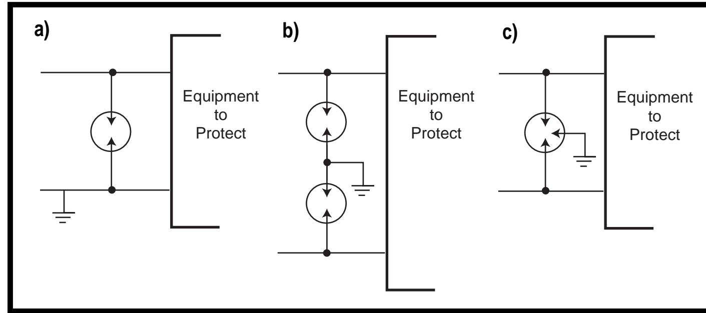


Figure 5a—*To protect a single-wire system, use a two-electrode gas tube.* ***b—****For a two-wire system with two two-electrode gas tubes, you have an extended turn-on time.* ***c—****A three-electrode gas tube is a better solution for two-wire systems.*

When the voltage across the tube reaches the breakover limit, the device begins to conduct. For a current less than 1 A, the gas-tube device is in the glow mode and the voltage across the device ranges from 50 to 150 V.

When the current through the device is greater than 1 A, the device is in arc voltage mode. In this mode, the current through the device can be several thousand amperes and the voltage across the device is in the 10–30-V range, as illustrated in Figure 4.

Once the surge dies off, the surge arrester returns to its high-impedance state only after the current through the devices goes below I_{Hold}. Notice the similarity between the semiconductor crowbar device in Figure 3 and the arcing device.

You should be aware that there are both two- and three-electrode devices. Two-electrode devices protect single lines, and three-electrode devices protect two-wire systems.

When a common-mode surge appears on a two-wire system, as in Figure 5, the surge travels down each line at a different rate due to the difference in line impedance. If two two-electrode devices are used (see Figure 5b), one device turns on sooner than the other, resulting in an overlap of

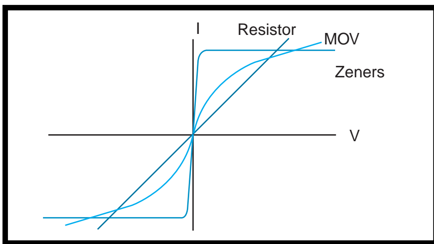


Figure 6—*Here's a VI curve for a MOV, back-to-back zener diodes, and a resistor. Notice how nonlinear the MOV's VI curve is when compared to the resistor. Also notice that the MOV's turn-on is soft, as compared to the hard turn-on of the zener diodes.*

turn-on delays and an extended turn-on time.

In the three-electrode device, the first surge ionizes all the gas in the device. Thus, when the second surge arrives, it is diverted to ground with no delay.

With gas-tube arresters, there are two things to be careful with. First, the current through the device must be reduced below the glow current or the device will "hold" in the glow state. Normally, when the surge dies out, so does the surge current, but you should still watch out for the hold phenomenon.

Another thing to keep your eye on is the *dv/dt* of the incoming transient. A finite time is required to ionize particles between the electrodes. Generally speaking, the device turns on in less than 1 µs. Therefore, faster transients can exceed the gas tube's breakover voltage momentarily.

So, if a gas arrester with a breakover voltage of 100 V and a turn-on time of 0.5 µs is subjected to a transient with a *dv/dt* of 1 kV/µs, the arrester breaks over at ~500 V, not the specified 100 V. If the same device is subjected to a 10-kV/µs transient, it strikes at ~5000 V.

The main advantage of the gas tube is its high-current-handling capability. Usually, gas tubes are employed as the first line of defense, being placed at the entry points of a piece of equipment to be protected. Secondary protection is normally required since gas-tube breakover voltages are in the 80–1000-V range.

Another advantage of gas tubes is that they have long lifetimes and require no maintenance, unlike air and carbon spark gaps.

METAL OXIDE VARISTORS

A MOV is a voltage-dependent resistor with a nonlinear VI curve. Varistors are monolithic devices consisting of many grains of zinc oxide combined with small amounts of metals (e.g., bismuth, cobalt, manganese, and other metal oxides).

The mixture is compressed into a single form. The result is a matrix of zinc-oxide grains that provide back-to-back PN-junction diode characteristics.

When the MOV is exposed to a surge, it behaves as an array of series and parallel connected diodes. This behavior results in the voltage across the MOV being clamped and the surge current being absorbed.

Figure 6 shows the VI curve of a resistor, MOV, and back-to-back zener diodes. Notice the nonlinear VI curve of the MOV with respect to the resistor. Also notice that the clamping action of the MOV is softer than that of the zener diode as mentioned previously.

However, a MOV absorbs much more energy than a zener diode. This is due to the fact that the MOV's ability to absorb energy depends on the amount of material present, whereas the zener's ability to absorb energy depends on the size of its junction area.

MOVs are two-terminal devices, with one terminal connected to the line being protected and the other terminal typically connected to ground. When the applied voltage is below the breakover voltage, the MOV appears as a high-impedance device with a leakage current in the range of 5–250 µA and a capacitance of 10–10,000 pF.

When the voltage across the MOV reaches the breakover and clamping voltages, the MOV goes into its low-impedance state. In this state, the MOV clamps the voltage across, diverting the surge current away from the line being protected.

When the voltage across the MOV goes below the breakover voltage, the MOV returns to its high-impedance state. There is no hold current as in crowbar devices. MOVs' clamping voltages range from about 6 V to several kilovolts, and their turn-on time is in the 50-ns range.

MOVs absorb the transient energy and dissipate it as heat, like all resistors. Therefore, a MOV can handle a finite amount of energy, given by the MOV's joule rating. The joule rating is normally specified for one pulse of a standard waveform:

$$E = KVcIp$$

where E is the energy the MOV can absorb, K is a constant (i.e., 1 for a rectangular and 8/20 waveform and 1.4 for a 10/1000 waveform), and Ip and Vc are the peak current and clamping voltage, respectively.

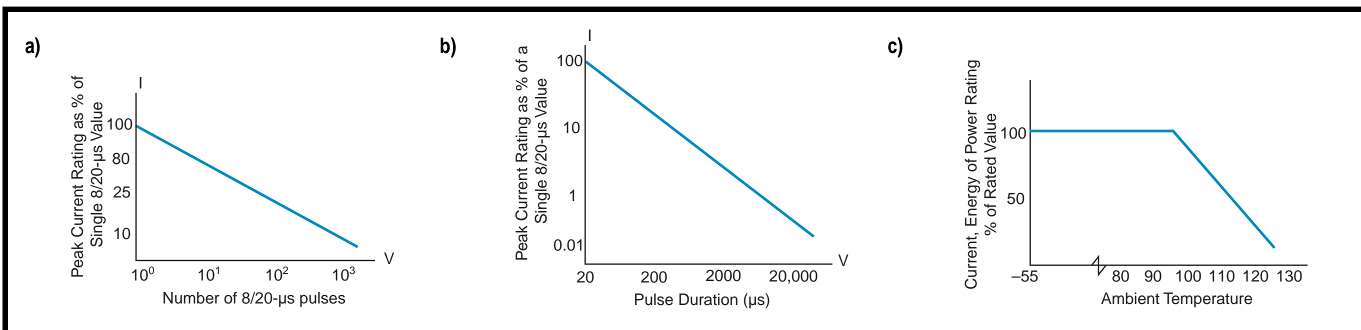


Figure 7a—*As the number of pulses a MOV is subjected to increases, M_{cross}, the MOV's current handling capability, measured as a percentage of its single pulse current handling ability goes down.* ***b—****As the pulse duration increases, the peak-current handling capabilities of the MOV decrease.* ***c—****As the ambient temperature passes a threshold, the ratings of a MOV must be reduced.*

Manufacturers typically supply curves that show how the MOV behaves when subjected to a series of transient pulses (see Figure 7a). As you'd expect, as the number of pulses increases, the energy-handling ability decreases.

Another useful curve that manufacturers provide is the maximum current versus pulse width shown in Figure 7b. Again as expected, the longer the pulse width, the lower the peak current.

Of course, if the pulse width is less than an 8/20 waveform, then the MOV could handle a current larger than *Ip*. Regardless of whether the MOV handles one transient or a repetitive set of transient pulses, the power rating of the MOV must be derated to account for ambient temperature (see Figure 7c).

Every time a MOV clamps a transient, it degrades slightly. This degradation is due to a small percentage of the device's internal diodes fusing and becoming permanently shorted.

The result is an aging effect with respect to the number of transients absorbed, increasing the leakage current. Also, if a MOV is subjected to a large current spike for an extended period, all of the device's diodes permanently short. Therefore, it's a good idea to fuse MOVs.

MOVs are commonly used in AC applications in conjunction with arc-type suppressers. The MOVs don't have the current-handling ability of the gas tube, but they turn on much faster and reduce the voltage overshoot associated with arc-type devices.

Due to their high capacitance, MOVs aren't used in high-speed circuits. Their high capacitance coupled with lead inductance works to form a low-pass filter. Surface-mount and leadless MOVs are available, but for applications greater than a 100 kHz, I tend to use other devices.

MORE TO COME

There are a lot more types of surge-suppression devices you need to consider. So, join me next month as I take a look at zeners, TVS thyristors, and diodes. ◼

Joe DiBartolomeo, P. Eng., has over 15 years' engineering experience. He currently works for Sensors and Software and also runs his own consulting company, Northern Engineering Associates. You may reach Joe at jdb.nea@sympatico.ca or by telephone at (905) 624-8909.

REFERENCES

Harris, *Transient voltage-suppression handbook*, 1994.

KeyTek, Surge-protection test handbook, 1986.

J. King, "Comm systems need protection from lightning," *EE Times*, February, **92**, 1997.

MAIDA, Zinc-oxide varistors for surge protection.

MTL, Surge-Protection App note, 1993–1994.

MTL, App. note AN9009, 1990.

I R S

425 Very Useful
426 Moderately Useful
427 Not Useful

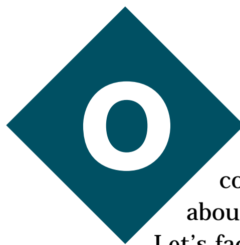
Proprietary Serial Protocols

FROM THE BENCH

Jeff Bachiochi

No Help from Traditional UARTs

Even when you don't have the tools, you need to know how to get the job done. For instance, with proprietary protocol message formats, traditional UARTs aren't an option. Jeff checks your options.



One of my favorite commercials is about automobiles. Let's face it: next to our homes, our autos are our largest investment. We pay more for auto maintenance than health care.

Anyway, Joe Customer drives into a service center to get a new battery installed in his vehicle. The scene opens with two mechanics under the hood.

Protruding above everything else in the front-most corner is an oversized battery. Joe asks, "Isn't that battery too large?" One mechanic answers, "No problem. We'll make it fit."

The scene flashes back to two boys standing behind a table. One is holding a rather large sledgehammer and is proudly grinning. His brother praises him saying, "Good job."

On the table is a child's shape toy that has a number of different-shaped holes. In the center triangular hole is a round wooden peg beaten into submission.

I still chuckle when I think about the commercial. On the flip side, did you know that was an actual technique used to make wooden dowels in many woodworking shops?

The point is, we don't always have the right tool for the job for whatever the reason. Can you accomplish the task even when your tools fail?

TRADITIONAL UARTs

Speaking of tools, almost everyone has worked with a Universal Asynchronous Receiver Transmitter (UART). The most common is the hardware UART.

The UART is capable of translating a serial bitstream to and from a parallel word. It was originally developed to provide a cost savings in copper when transmitting data over long distances. Although the data traveling single file through one wire is less than one-eighth the data rate of a parallel eight-wire transfer, the copper savings was worth the speed penalty.

However, for the serial data caught by the receiver to be recognized as the same data sent by the transmitter, the transmitting and receiving UARTs must play by the same rules.

RULES

Probably the most important rule in serial data transmission is the bit timing. Beginning with RTTY (radio-teletype), the operating speeds of most mechanical machines (like those used by Western Union) sending Baudot code were on the order of 60–100 wpm (or 6–10 characters per second). Or, as we call it, baud rate.

Unlike human speech where you can probably continue to understand the conversation independent of the speaker's pace, UARTs must talk and listen at the same baud rate. This task is usually accomplished by a combi-

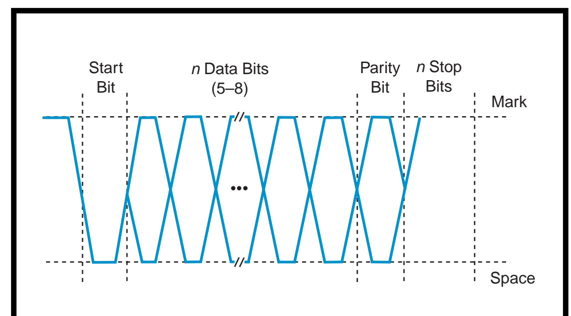


Figure 1—*All UARTs require a start bit (space state) followed by data bits, an optional parity bit, and at least one stop bit (mark state).*

nation of hardware and software design.

Some oscillating standard (e.g., a crystal or other clock/clocking device) is input to the UART (a UART built into a micro may use the micro's master clock). This UART clock usually goes through a software-programmable divider to enable the UART's bit rate generator to be adjusted to one of many standard baud-rate values. Now, the transmitting and receiving UARTs can divide time into identical-length time slots.

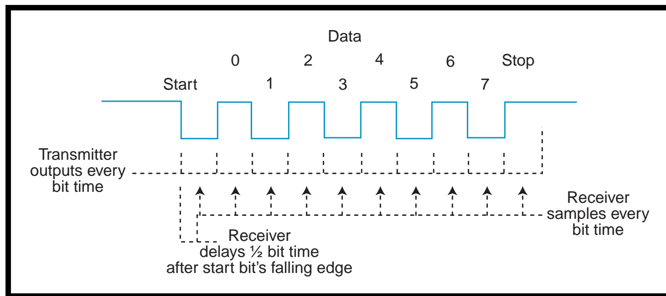


Figure 2—*For two UARTs to communicate, they must change and sample bits only at prearranged times.*

Since both UARTs have free running bit-rate generators, the next rule ensures that the two UARTs stay in sync with each other. The transmitter output is a digital value, so it can be in only one of two states—a logic 1, called the mark state, or a logic 0, known as the space state. When the transmitter is not sending data, it must remain in the mark state.

To get the receiver's attention and consequently sync its bit-time generator with that of the transmitter, the receiver sends out a start bit. The start bit is always a space equal to one bit time. When the receiver sees the falling edge, it restarts its bit-time generator.

Next, we get to the actual transmission of data. The maximum number of bits in a byte is eight, but that doesn't necessarily mean that a data word has eight bits. The early RTTY Baudot code was only five bits long. ASCII data is only seven bits long.

The data word length (i.e., the number of data bits actually sent) must be selected to be the same for both the transmitting and receiving UARTs. Typically, it's between five and eight bits.

It also makes a difference whether the data bits are transmitted least or most significant bit first. UARTs use the least significant bit-first convention. Each data bit is always equal to one bit time.

To add some security to the transmission, UARTs have an optional parity bit. The UART must be told whether this parity bit is used. If it is used, there are a number of options about how it is implemented.

The parity bit can always be either a mark or a space, or it can be based on the data bits. Parity that is based on the data bits can be either odd or even.

Odd parity defines the parity bit as whatever level is necessary to make the total number of 1 bits (including data bits and parity bit) odd (i.e., 0110111 + ? = 1). Even parity is defined as the level needed to make the total number of 1 bits even (i.e., 0110111 + ? = 0). The parity bit is always equal to one bit time.

Finally, to signify the end of the single character transmission, at least one stop bit is sent. A stop bit is always a mark equal to one bit time. A transmission must have at least one stop bit, but the UART can usually be set to transmit one or more stop bits. Extra stop bits give the receiver a little more time to get ready for the next start bit.

The receiving UART can be set to fewer stop bits than the transmitting UART because this time is essentially idle. (Note: if one UART forces you to use one kind of parity which the other

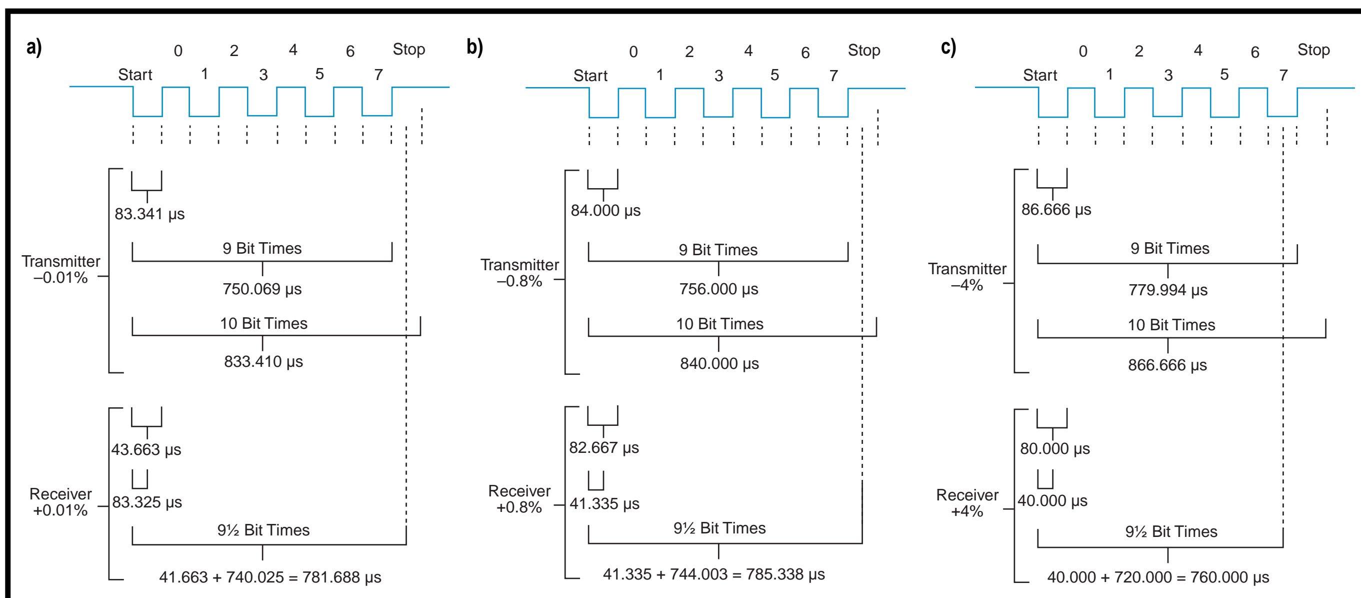


Figure 3—*These diagrams show how tolerance can affect the data received by inaccurate bit timing for UARTs when transmitters are on the slow side of the tolerance and the receivers are on the fast side of the clock's tolerance. For a UART using a crystal (**a**), the nominal time (what the transmitter is using) for 9½ bits is 792 µs (83.341 × 9.5), but the receiver's actual time is 781 µs. For a UART using a resonator (**b**), the nominal time for 9½ bits is 798 µs (84 × 9.5), but the receiver's actual time is 785 µs. For a UART using a internal RC oscillator (**c**), the nominal time for 9½ bits is 823 µs (86.666 × 9.5), but the receiver's actual time is 760 µs, which is not within one bit time.*

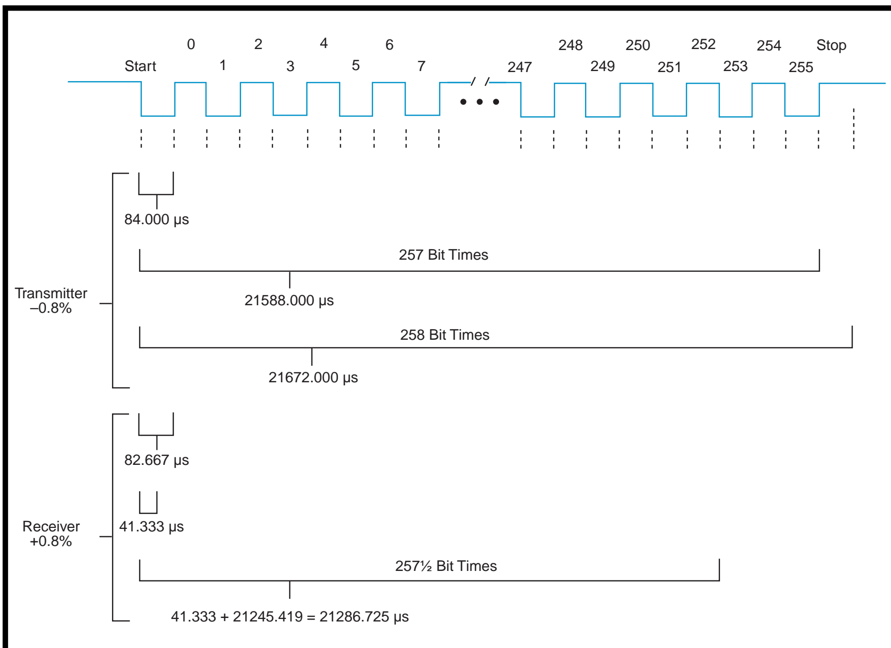


Figure 4—*Using a resonator for the UARTs would mean that at the end of the transmission, the receiver would be off as much as four bit times, which is totally unacceptable.*

doesn't support, mark parity looks just like a stop bit.)

Although not previously stated, the entire character transmission, including start plus data plus (parity) plus stop, must be sent sequentially using consecutive bit timing (see Figure 1). The whole sequence begins when the data to be transmitted is written into the transmit register.

The UART handles the timing and bit output of the complete character transmission, leaving the program free to handle other tasks. An interrupt can indicate when the transmitter UART is free for the next character. In fact, some UARTs have a buffer that can hold a number of characters to be transmitted.

On the reception side, the receiving UART automatically synchronizes its bit timing by delaying a ½ bit time after the falling edge of the start bit. (To minimize the delay in synchronization, the receiver must have edge-detection circuitry or sample the input at a fast rate. Most sampling rates are 16× the bit time.)

After the initial ½ bit time, which offsets the bit sampling point into the (assumed) center of the transmitted bit time, sampling the incoming bitstream after each full bit time enables the data word to be reformed by shifting the samples into a receive register.

In addition to sampling for data, the receive UART also tests and calculates parity (if used) and reports any errors in the status register. A framing error can be generated by the receiving UART if the stop bit was not received correctly. The framing error indicates the data is probably invalid due to noise within the transmission or a data set using an incompatible data (bit) rate.

If a second character is received before the receive register has been read (by the executing program), an overrun error is flagged. This flag indicates some data has been lost because it's coming in faster than it's being processed.

So, what can the UART do about these errors? Absolutely nothing. It's up to the executing program to institute some kind of error correction, possibly by asking for the information to be retransmitted, but this goes beyond the scope of this article.

WHEN WHAT YOU'VE GOT WON'T DO

If you wish to remain compatible with the world's serial communication standards, you must choose a protocol (i.e., a data word format) that fits in with the mainstream. The most widely used protocol is probably 8N1—1 start bit (assumed), 8 data bits, no parity (not used), and 1 stop bit.

But, what happens when the protocol you need to be compatible with falls outside the normal standards? The hardware UART, based on the fixed set of rules, becomes unusable.

Recently, an application surfaced in which OEM equipment in a distributed control system needed a redesign. Intersystem communication, which used an existing proprietary protocol message format, had to remain intact.

The protocol used a 256N1 format. So, hardware UARTs couldn't be used. Enter the software UART.

SOFTWARE TO THE RESCUE

Implementing a software UART is not a momentous task. But, it does require a bit more processing time.

The most important routine is implementing some kind of bit-timing strategy. If you have a reload timer available, you can initialize it to reload itself with a value that overflows on exactly one bit time.

If the reload function isn't available, your code must pay attention to the overflow and manually reload it with a bit-time value adjusting for the execution time your code takes to acknowledge the overflow and reload the timer. The worst scenario requires you to actually count cycles between outputting serial bits because you don't have a timer.

The accuracy of the communications depends on the clocking source. Not only is using a crystal necessary, but choosing the right speed is of extreme importance.

The crystal frequency should be selected such that the timer's overflow occurs at exactly the prescribed bit rate. This usually means using a crystal whose frequency is an even multiple of the selected bit rate.

On the transmitting side, you can't just pop your data into a transmit register and go away until the hardware has done its job. Your code becomes responsible for outputting a start bit, the required number of data bits, calculating the parity bit (if necessary), and finishing with the appropriate number of stop bits.

If the timer is available, your program can go off and do some other processing after you've set the output

bit appropriately and are waiting for the bit time to expire.

When using high baud rates, you may not have time to go off and do other processing. You may have to remain with the communication routine until you have completed the whole task.

If a timer isn't available, you are stuck looping until you execute the number of instructions that equals a bit time because you must ensure that the count remains accurate.

On the reception side, the reconstruction of the data must be handled by your code following the same techniques as the hardware UART. Again timing is most important. Since you never know when a start bit may come, your code must rely on an interrupt or continuous polling.

The most favorable time to sample the input for data is during the center of the transmitted bit time. Calculating this (½ bit time) point is based on when the receiver first sees the start bit's falling edge.

Once this estimate is made, successive whole bit-time samples are used to reconstruct the data. Error checking and received data processing must all take place prior to the beginning of a new character transmission.

TIMING

Let's take a look at how the transmitter and receiver tolerances affect communication integrity.

The standard AT-cut microprocessor crystal is ±100 ppm (or 0.01%) over the 0–70°C temperature range. A ceramic resonator is about ± 0.8% over the same temperature range.

A particular micro using an internally trimmed R/C oscillator could be off as much as 6.25% over the same temperature range, while an external R/C could be off much more, depending on the tolerances of the two parts.

Figure 2 shows a nominal 8N1 serial transmission with nominal reception timing.

Figures 3a–c show transmitters with timing on the lower limits of the tolerances and reception sampling based on the upper limits for each of the three clocking sources—crystal, resonator, and internal R/C.

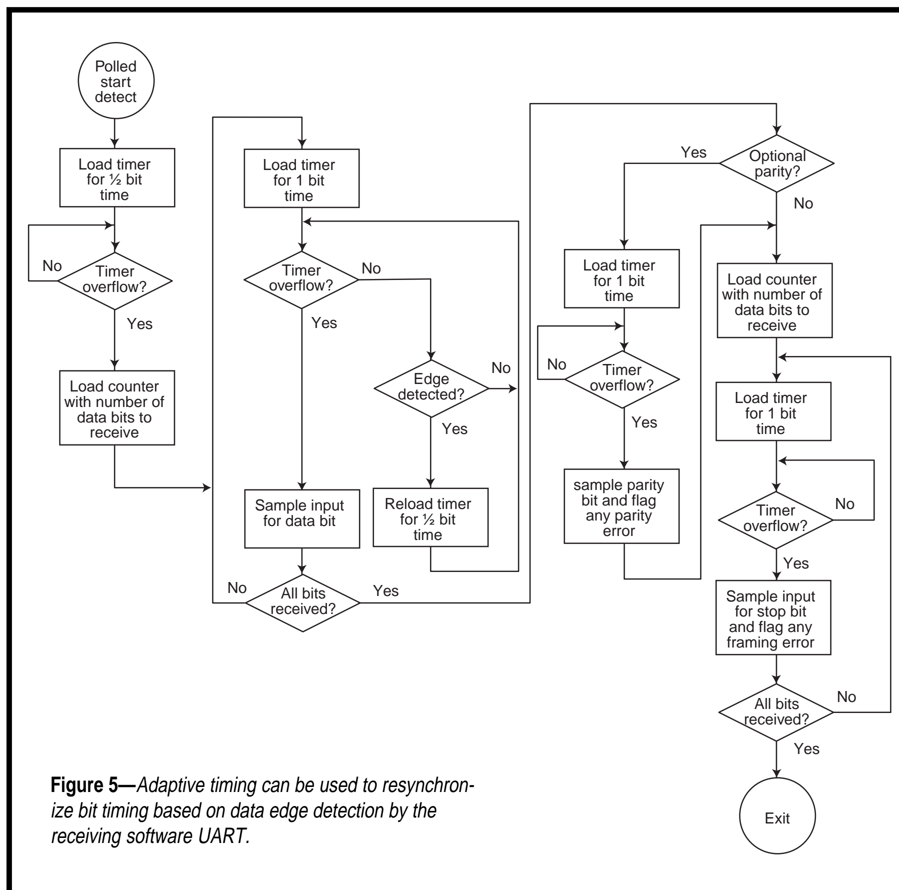


Figure 5—*Adaptive timing can be used to resynchronize bit timing based on data edge detection by the receiving software UART.*

In many designs, not much thought is given to baud-rate divisors. Crystal frequencies are often picked for maximum speed of execution. Baud-rate tables located in processor manuals usually give a list of baud rate and accuracy breakdowns for various crystal frequencies.

In this example, a deviation of a few percent wasn't a problem for most UARTs, as this was well within the operating tolerance of the device.

But, if the clocking frequency is not an even multiple of the baud rate, the timer overflow can never be on the mark (so to speak).

If this is the case, you start out with an inaccuracy and it can get much worse from there. As the total tolerance excursions approach 10%, the 8N1 protocol approaches doom. You must also take into account things like the receiver's lag time on start detection or the slew rate of the data edge.

Now imagine extending the data word out to 256 bits instead of just 8. System tolerances must be held extremely small or accurate communication cannot be achieved.

Figure 4 shows the same tolerance picture as the 8N1 protocol. This time, however, it is extended out for the full 256 bits of the proprietary protocol we must comply with. All the sudden, even a resonator's clock accuracy doesn't look that good.

If a system was not designed with these parameters in mind, we could be in deep trouble. Fortunately, system designers are well-aware of the need for accurate baud-rate generation in order for their system to use this proprietary protocol.

This unusual communication protocol makes it most difficult for external equipment to listen in, which might be the reason for its origin. It may have been designed for efficiency by some system's designer, but the poor software and hardware developers who had to make that dream a reality must have sought a gruesome revenge.

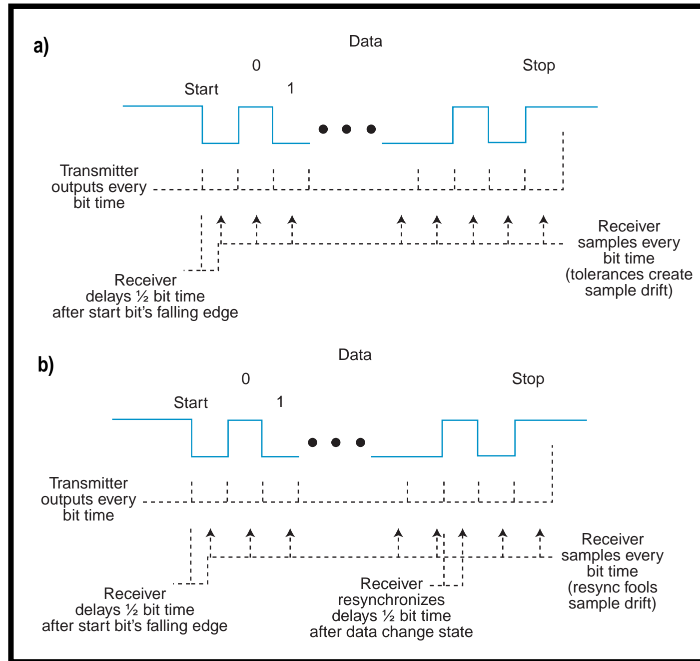


Figure 6—*Here you can see the difference between nominal receiver timing (**a**) and creeping receiver timing (**b**) which has been corrected by data.*

ADAPTIVE TIMING

If the system design doesn't have the accuracy necessary, is there any hope of implementing this type of protocol? Yes and no.

It depends on the data being transmitted. In a large packet like this proprietary protocol, the data is likely transmitted in some kind of limited set that's not a binary transmission.

Since binary transmissions can include data containing many 00s or FFs, and this protocol has only one start and one stop bit, it's possible that there are no data transitions throughout the entire packet, making adaptive timing ineffective.

Adaptive timing is based on the assumption that data changes states from time to time throughout the data-word transmission. If this assumption is valid, you can add adaptive timing to the receiver's code.

Simply put, if the data changes state within an expected window, you need to resynchronize the receiver's bit timer. This task is accomplished by reloading the timer with the appropriate value of ½ bit time no matter where it is in its counting cycle (see Figure 5). Figures 6a and b demonstrate how this solution can help the timer recover in an out-of-tolerance system.

One danger of this technique is noise on the transmission. If noise is detected as a legal transition and the timer synchronizes to it, sampling proceeds based on the noise transition, most likely giving erroneous data. Such are the tradeoffs.

DATA ACCURACY

On the whole, data accuracy can only be assured by using a well-designed protocol. To receive reliable data, start with an accurate transmitter and receiver. Then, add some kind of data check. The most common is the use of parity (and it's free with most UARTs).

In larger packets, like the 64-data-word–sized protocol I needed to design for, you can use checksum or CRC data checks built into the packets. It adds a bit of overhead to the receive routines to ensure accurate reception and also requires a method of requesting retransmission of a damaged data packet from the transmitter.

So, the next time you have a communication job to do, make use of that good old standard, the hardware UART. But if the bits get out of control, take over and bang 'em into submission.

Just remember to get out your slide rule and check the system tolerances—because the impossible is a wee bit more difficult. ■

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

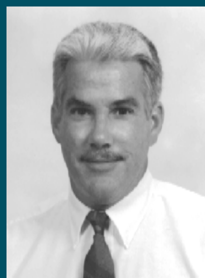
IRS

428 Very Useful
429 Moderately Useful
430 Not Useful

SILICON UPDATE

Tom Cantrell

ShBoom Box



A little bored? Patriot Scientific's ShBoom to the rescue. It brings past and present together with stack-machine architecture, Java portability, and code density to make one mean little CPU.

It's been said that inside every reporter is a Great American Novel. It's also been said that inside is where it should stay! So, while you won't find mine on the bookshelves anytime soon, it goes something like this.

Imagine a time when science eliminates death from natural causes, though you can still get taken out by unnatural causes. (In the sequel, they just grow a new you whenever the old one breaks.) Sounds grand, eh? Uh-uh, no way.

In fact, give evolution enough iterations under such a scenario and what'll be left won't be people but spineless, anxiety-ridden losers that scurry underground the day they leave the test tube. Think about it next time you hop into the minivan to make a run for tofu and diet soda. Better pick up some sunscreen and condoms, too.

How does this relate to chips, you may ask? Well, with the Silicon Wizards giving all we ask, I fear that the result is kind of boring—safe cars, safe diets, safe sex, and now, safe chips.

Of course, you can guess the final chapter of my book. A small tribe of untamed wild ones keep the spark of humanity alive and ultimately save the day.

Are any chips left with the passion and zest for life that's always been the heart and soul of high tech? Let's take a close look at a chip with a lot of spirit—Patriot Scientific's PSC1000 ShBoom CPU.

TROJAN CHIP

"XYZ company introduces their new high-performance, low-cost, and easy-to-use 32-bit embedded micro. With C, Web, and Java support, the chip is ideal for set-top boxes, PDAs, and office equipment. Benchmarks prove...."

That PR could apply to any number of chips, including, frankly, the PSC-1000. Even a cursory glance under the hood reveals few surprises, as you see in Figure 1.

The clock generator requires an oscillator input, which is PLL-boosted

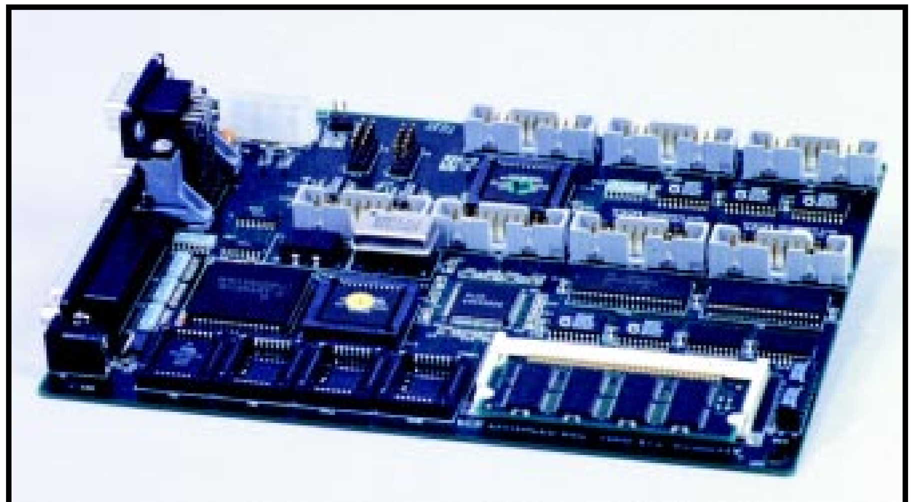


Photo 1—*The PSC1000 evaluation board includes plenty of memory (ROM, SRAM, and DRAM), PC-like (2S+P) I/O, and debugging/expansion headers.*

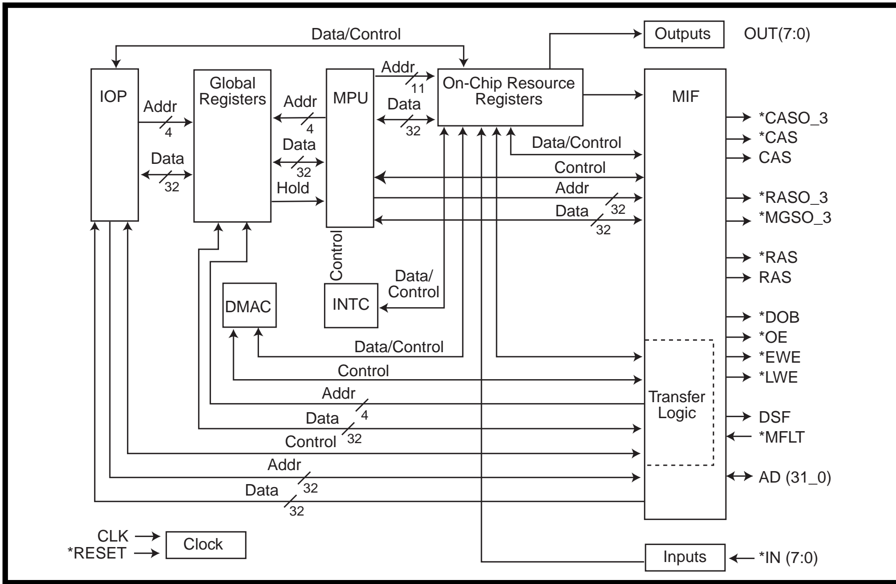


Figure 1—*The PSC1000 memory interface (MIF, which supports direct DRAM connection), interrupt controller (INTC), and DMA controller (DMAC) appear typical. It's the on-chip I/O coprocessor (IOP) and, most of all, what's buried inside the MPU block that make the chip unique.*

internally 2× to clock the MPU and 4× for fine-resolution bus timing. A four-bank memory interface (MIF) accommodates various combinations, widths, and speeds of ROM/EPROM, SRAM, DRAM, and VRAM.

The eight-channel DMAC includes bus-matching support for byte, four-byte, and cell (32 bit) transfers. Eight bits each of input and output can be configured as control signals for the DMAC or interrupt inputs in addition to general-purpose I/O.

Although the concept of including a separate I/O processor (IOP) isn't new, the implementation is somewhat novel. First, instead of a dedicated memory, the IOP fetches instructions externally via the MIF, contending for access with the MPU and DMAC. Communication with the MPU (and DMAC) is accomplished via 16 global registers (see Figure 2) accessible to all.

Reflecting the real-world time constraints imposed on I/O, the IOP is given top priority. Thus, the most important instruction in the IOP's minimal (12 instruction) repertoire is `DELAY`, which puts the IOP to sleep for a particular amount of time, relinquishing the MIF to the MPU.

In fact, every time the MPU (and DMAC) requests access to the MIF, a slot check is performed to guarantee there's time to complete the requested transaction before the IOP comes out of `DELAY`. Such deterministic scheduling is possible because the details of bus timing are completely known internally. The emphasis on no ifs, ands, or buts I/O timing goes so far as to preclude an external WAIT input and its accompanying temporal uncertainty.

The 100-pin (PQFP) chip runs at 3–5 V and provides separate power connections for the core, control signals, and the A/D bus. The current drive on key signal groups (i.e., RAS/CAS, control lines, A/D bus) is programmable. Using the minimum drive required by a particular design reduces the output edge rates, which cuts noise emissions.

FORTH TO THE PAST

Peering more closely at the innocuous-sounding MPU block reveals a chip that marches to a different drummer. As shown in Figure 3, the PSC1000 architecture is atypical, incorporating aspects of what old-timers might recognize as a stack machine.

Goethe said something like, "Everything has been thought of before, but the problem is to think of it again." And, it's true in this case as well.

In fact, stack machines have a proud tradition dating to practically the dawn of computing. For instance, back in the '60s when the only computers were mainframes, a company called Burroughs designed the innovative stack-oriented B5000. Although Burroughs, like a bunch of other would-be competitors, faded under the mainframe hegemony of IBM, interest in stack machines continued to grow.

The Golden Age of the concept was ushered in with the invention of the Forth language in the mid '70s by Charles Moore and Elizabeth Rather [1]. Reflecting the starry-eyed faith of the inventors, the first applications were controlling the giant telescope at Kitt Peak National Observatory.

I myself did more than a bit of fooling around with Forth, which offered a number of unique advantages including economy, performance, interactivity, and portability.

Remember, machines at the time were laughably limited. I was running a mighty 4-MHz Z80 with 64 KB of RAM, but even the minicomputers (e.g., the PDP-11) and mainframes (e.g., 360) of the time couldn't match today's PC. Effectively, the only programming options for me were ASM and BASIC.

Performance and economy were derived from the fact that Forth mapped naturally to minimalist hardware (i.e., a stack-oriented language for stack machines). The PSC1000 lineage is easily discernible in papers

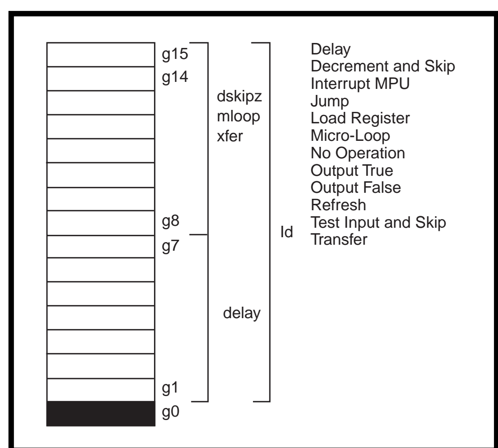


Figure 2—*The general registers serve as the link between the MPU and IOP. The IOP instruction set targets real-time I/O and is extremely reduced. The MPU can only run when the IOP is executing* `DELAY` *(i.e., I/O has the highest priority).*

Photo 2—*Though most of the DOS-based development software is command-line driven, the debug front-end includes a simple GUI. Notice the grouping of byte-wide instructions into 32-bit cells in the assembly-language window.*

and articles of the time describing homebrewed Forth engines [2].

As much as the language itself, implementation as an interpreter was key. This meant Forth memory needs were low both for development and at run time. Development was also highly interactive, thanks to elimination of cumbersome compile and link steps.

The combination of a simple machine model and interpretive execution meant Forth could be, and was, easily ported to many different machines. All that was required was a few kilobytes to implement a virtual stack machine and seed the dictionary with a basic vocabulary. You'd use these words to build your own words in a hierarchical manner, hiding complexity along the way.

It was fun, but eventually the party was over. For all its niceties, Forth suffered from some flaws.

Most apparent was the RPN notation intrinsic to the stack concept. Instead of writing $(A \times X) + B$, you entered $A\ X \times B\ +$. Despite the fact that scientists might actually prefer the elegance of RPN, it made for dubious readability.

Another problem was that stacks, although great for calculations, are quite limiting in other ways. Invariably, much head scratching revolved around the need to get at some deeply buried element, to which end a variety of stack manipulations (e.g., duplicate, swap, rotate elements, etc.) were required.

Ultimately, macromarket forces caused Forth to fade. The appearance of the PC changed the rules, the issue becoming whether any other architecture—not to mention an unconventional one—could survive.

The subsequent explosion in computing capabilities rendered the effi-

ciency issue somewhat moot. After a bit of flirting with Forth and other languages like Pascal and Ada, the programming community decided to hitch up with C.

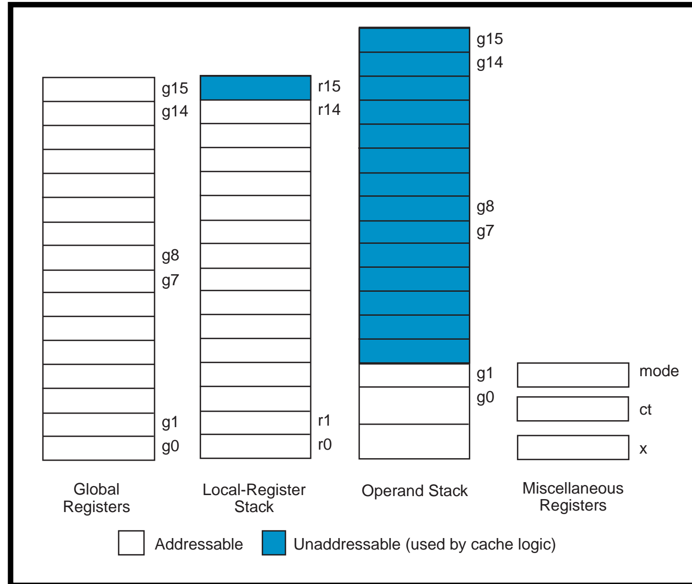


Figure 3—*The PSC1000 is a hybrid register/stack architecture that tempers Forth roots with register reality. Most instructions (ALU ops and loads/stores) work on the top of the operand stack, but data can be moved to local and global registers as well. x is a dedicated index register (s0 and r0 also work as indexes), and ct a loop counter. All registers are 32 bits wide.*

BACK TO THE FUTURE

So, why sift through the history books today? Well, remember Goethe. Just because an idea came and went doesn't mean its time won't come again.

For instance, doesn't the idea of running an interpreted language on a virtual machine to achieve true portability sound familiar? Have a cup of coffee while you think about it.

Naturally, the company exploits the linkage with the Java craze. They've licensed the required technology from Sun, are working on their virtual machine, and expect competitive Caffeinemarks.

However, I suspect the ultimate fate of Java—whether it takes off and what chips it runs on—is as likely to be decided in a courtroom than in the lab.

In the meantime, the PSC1000 is certainly competitive in traditional embedded applications. After all, the most popular embedded micros ('51, '68, PIC, '*x*86, etc.) aren't exactly spring chickens themselves.

And, the embedded market still cares about things like code density—a stack machine forte. Consider the PSC1000 instruction set in Table 1. It has a number of interesting features, but a real standout is the fact most instructions are a measly byte long.

As shown in Figure 4, and not unexpected, the only exceptions are branches and literals. Both take advantage of a four-instruction (i.e., the 32-bit bus width) grouping concept to expand opcodes as necessary.

Grouping also supports the concept of micro-loops (i.e., loops that fit entirely within a group). In this case, the group is buffered on-chip and acts as a minicache, speeding access and freeing the internal bus.

The addition of registers eases the dreaded stack bottleneck. Yes, all ALU ops and loads/stores work with the operand stack, and there are even stack-shuffling instructions like `EXCHANGE` and `REVOLVE`, but it's easy to move data to and from registers as well.

The local registers can either be accessed as a stack (e.g., return addresses) or directly (four-bit register number in opcode). The global registers are only directly accessed, reflecting their primary role as interconnect with the IOP and DMAC.

Most of the simple ALU ops execute in a single cycle of the 2× clock (i.e., 50 MIPS with a 25-MHz oscillator). Numeric operations are slower (e.g., multiply and divide are 32 clocks) but supplemented with a selection of housekeeping aids for floating point. Of course, instructions that require memory accesses (e.g., branches, loads, and stores) depend on external bus timing.

Along with micro-loops, another small concession to caching is 16-

Arithmetic/Shift
ADD
ADD with carry
ADD ADDRESS
SUBTRACT
SUBTRACT with borrow
INCREMENT
DECREMENT
NEGATE
SIGN EXTEND BYTE
COMPARE
MAXIMUM
MULTIPLY SIGNED
MULTIPLY UNSIGNED
FAST MULTIPLY SIGNED
DIVIDE UNSIGNED
SHIFT LEFT/RIGHT
DOUBLE SHIFT LEFT/RIGHT
INVERT CARRY

Floating Point
TEST EXPONENT
EXTRACT EXPONENT
EXTRACT SIGNIFICAND
RESTORE EXPONENT
DENORMALIZE
NORMALIZE RIGHT/LEFT
EXPONENT DIFFERENCE
ADD EXPONENTS
SUBTRACT EXPONENTS
ROUND

Miscellaneous
CACHE CONTROL
FRAME CONTROL
STACK DEPTH
NO OPERATION
ENABLE/DISABLE INTERRUPTS

Control Transfer
BRANCH
BRANCH ON ZERO
BRANCH INDIRECT
CALL
CALL INDIRECT
DECREMENT AND BRANCH
SKIP
SKIP ON CONDITION
MICRO-LOOP
MICRO-LOOP ON CONDITION
RETURN
RETURN FROM INTERRUPT

Logical
AND
OR
XOR
NOT AND
TEST BYTES
EQUAL ZERO

Data Management
LOAD
STORE
STORE INDIRECT, pre-dec/post-inc
PUSH REGISTER/STACK
POP REGISTER/STACK
EXCHANGE
REVOLVE
SPLIT
REPLACE BYTE
PUSH LITERAL
STORE ON-CHIP RESOURCE
LOAD ON-CHIP RESOURCE

Debugging
STEP
BREAKPOINT

Table 1—*The instruction set is an interesting combination of RISC and CISC. The conventional ALU, branch, and load/store instructions are supplemented with stack-centric ops and floating-point assists.*

entry storage for the operand and local stacks. On-chip hardware automatically spills and refills as stack accesses cross the boundary.

Since cache effects can compromise determinism, CACHE-CONTROL instructions give explicit control to those who need it. The DEPTH instruction reports how many items can be removed from a stack without causing a refill, while the CACHE instruction prepares the stack to accept or deliver a specified number of operands without interruption.

LINGUA FRACTION

The $299 evaluation kit I checked out comes with a board (see Photo 1), power supply, cables and a complete selection of PC-based development software (see Photo 2), including a C compiler. By the time you read this, you can contact the company for the latest status on Java and, yes, even Forth.

The board accommodates 4-MB DRAM, up to 1-MB SRAM, and up to 2 MB of flash memory. An additional DIMM socket lets you put on an additional 16-MB DRAM, while a 16550 serves up PC-compatible serial and parallel I/O ports.

Expansion headers make all critical signals accessible—a must, given the preponderance of impossible-to-probe fine-pitch surface-mount chips. Operation revolves around the usual compile, download, and debug ritual under control of a simple ROM monitor.

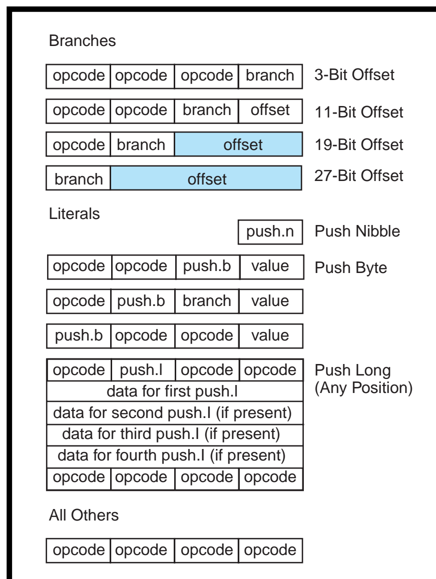


Figure 4—*The combination of 8-bit opcodes and 32-bit bus width lends itself to a four-instruction grouping concept. For instance, branch offset can consume a small portion (3 bits) or almost all (27 bits) of a group. Byte literals always occupy the last byte of a group, while long literals occupy their own group.*

Listing 1—*This C program computes the value of π.*

```
#include <stdio.h>
#include <math.h>
#define ITERATIONS 100000
int main(void)
{
  long i;
  int sign = 0;
  double pi = 0.0;
  printf("\n");
  for(i = 1; i < ITERATIONS; i += 2)
    pi += (sign ^= 1) ? 4. / i : -4. / i;
  pi += 2. / (ITERATIONS - 2);
  printf("pi is approximately equal to %.12f (%.12f)\n",
         pi, 4. * atan(1.0));
  return 0;
}
```

The package I received was definitely beta and a bit rough around the edges. A few incomplete docs, cut and jumps on the board, some finicky software, and such. Nothing that proved insurmountable.

While I'm sure the package will get fine-tuned, these tools will never win the Barney award. Running under DOS, there's little attention to cosmetics, IDEs, GUIs, and so on.

Instead, the software is of the traditional command-line power user sort. In other words, it works great once you get fully up to speed, but there's a lot of documentation to wade through.

Fortunately, a simple tutorial section steps through the compile, link, hex format, download, and run process. So, I tried the example in Listing 1, which exercises floating point to compute an approximation of π.

I then ran the same program on my Mac (16-MHz '030) but only after changing the `int i` in the original to a `long i` on the Mac. The Mac didn't complain about being asked to compare an `int` with 100000. It just never found a match.

For what it's worth, the PSC1000 with its 20-MHz oscillator (i.e., 40-MHz CPU clock) was about three times faster than my 16-MHz Mac (i.e., ~1.5–2 s vs. ~6 s). The code was substantially smaller as well (19 vs. 26 KB).

The results are certainly interesting and arguably even intriguing. They may not prove compelling for those who prefer to lead safe, quiet lives.

But, sometimes don't you just wish you could swap the minivan and tofu for a Humvee and T-bone? ◘

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.

REFERENCES

[1] C. Moore and E. Rather, "The Forth program for spectral line observing," *Proceedings IEEE*, **61**, September, 1973.

[2] J.C. Vaughan and R.L. Smith, The Design of a Forth Computer," *Journal of Forth Application and Research*, **2:1**, 1984.

SOURCE

PSC1000 ShBoom CPU
Patriot Scientific Corp.
10980 Via Frontera
San Diego, CA 92127
(619) 674-5000
Fax: (619) 674-5005
www.ptsc.com

I R S

431 Very Useful
432 Moderately Useful
433 Not Useful

PRIORITY INTERRUPT

For Once, I Sort of Agree

It's not often that I agree with Bill Gates, I assure you. From a technology viewpoint, we are worlds apart. My idea of computerizing something is a vision of making its operation simpler and more efficient. Every time I get involved in one of Bill's visions, I end up having to buy a new computer. Certainly, we don't debate that these offerings contain a modicum of enhancements and improvements. However, having to triple or quadruple the horsepower of your PC each time you upgrade the software leaves a lot to be desired. But don't worry, this isn't a tirade against Microsoft and I'm not going to reminisce about how much we used to do on an 8-bit processor with 64 KB.

Fact is, there's one issue where I might have to agree with Bill. In this latest face-off with the government, the makers of Netscape argue that a browser and an operating system are two separate things. It's OK to have customers buy your operating system, but to force them to all use your browser is monopolizing. Microsoft insists that there is no defining line between an operating system and browser. Supporting this opinion is the reality that a browser seems to be the user interface of choice in a majority of recently introduced software applications. Microsoft contends it is a natural evolution of technology.

I suspect that all those people who enjoy browsing the 'Net have a great deal to do with that evolution. It only takes visiting a few Web sites and executing a few online transactions to quickly realize that your browser is a universal entry vehicle into other systems. It gives you all the benefits of executing the online application without concern for the host's operating system or processor type.

The good news is that for many applications it offers a standard interface model. A remotely monitored refinery tank farm could have a unique communication protocol and a custom display medium. That would be the traditional approach. Today, however, it probably makes far more sense to design the monitoring system so it can interact with a browser. The user simply has to dial up the tank farm from anywhere with any computer and see what's going on.

There are clear advantages to using a browser as a front end for software applications. The user interface serves as an effective isolation between the user and the physical application hardware. Software changes and technical support need only be applied at the application end rather than to each user site. Want to expand the tank farm? Simply change the monitoring electronics and server software. The next time the user checks in, the browser shows 20 additional tanks. No fuss, no muss, no wiring.

The bad news is that there will be increased demand for everything being browser compatible. If we're not careful how it's done, browser-based closed-loop monitoring and control can become cutesy and inefficient. One of the things we have to be careful about in all this is that all this user interface and application isolation doesn't get out of hand. While it's easy to conclude that a browser makes an ideal user interface, I'm not all that convinced that enough thought is being given to the browser application itself. I don't write a lot of software, but I certainly believe that designing software for a browser application is significantly different than for a stand-alone operating system.

Someone suggested to me that there is a simple test to illustrate the obvious answer. Pick a dozen Web users at your office and look at their favorite-sites list. Invariably, Yahoo or Altavista, two of the 50+ search-engine sites, will appear on their list. If you ask why, most users simply say that it's because these sites are fast.

There's a natural tendency for developers to include fancy graphics, multiple windows, and lots of bells and whistles in their presentation pages. Yahoo and Altavista are fast because they avoid bandwidth-eating graphics and high-end features. We've all experienced the excruciating wait at Web sites that download page after page of useless, albeit flashy, graphics before they get to an index page. You could have breezed through a half dozen Yahoo pages in the same time.

Future implementation of browsers in embedded system applications is a given. Successful execution, however, is a careful balance between bandwidth and UI graphic necessity. I realize that the experience of the past suggests that one answer is to simply force us all to increase the bandwidth and computer horsepower once again. The other option is to put a little more thought behind this kind of software.

Yes, Bill, this is one of those occasions that I agree with you. Indeed, there isn't a clear line between browser and operating system anymore. Agreeing with you, however, doesn't mean that I'm willing to live with only one brand.

steve.ciarcia@circuitcellar.com