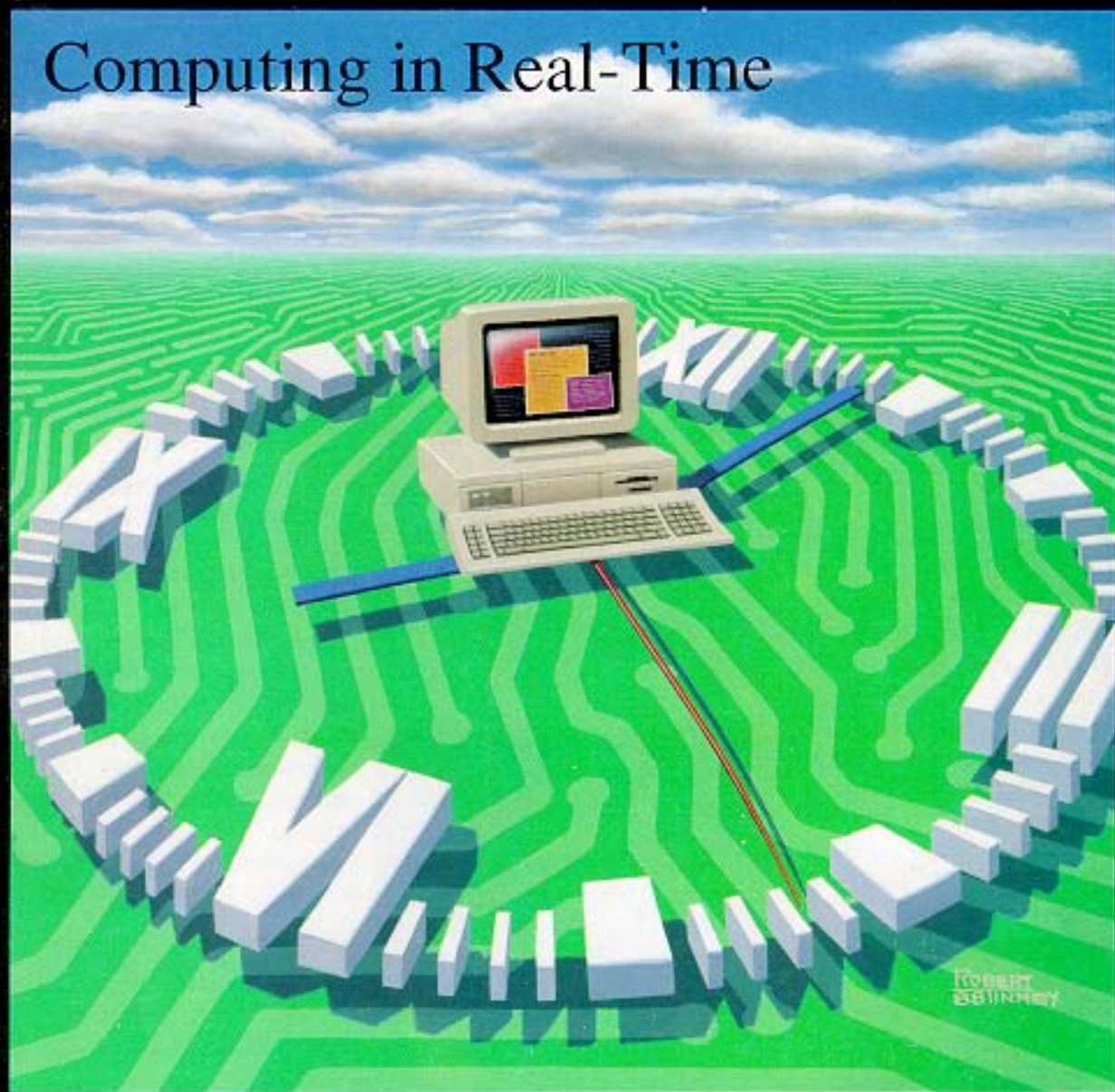


# Circuit CELLAR LINK<sup>®</sup>

THE COMPUTER APPLICATIONS JOURNAL

Computing in Real-Time



January/February 1989 — Issue 7

\$3.95

# EDITOR'S INK

---

## The Revolution Continues

As I write this, COMDEX is a two-week-old memory. I've had a chance to sit down and sift through the mountain of press releases, media kits, catalogs, and business cards that I brought back with me, and I've got to admit that I'm a little disturbed. Between networks, multiuser systems, and OS/2, the emphasis was almost entirely on (relatively) big computers and central control of programs and data. In this world, if you're not locked into a large project team at a multinational hardware or software manufacturer, you have no business mucking around inside the box. As a matter of fact, if you're trying to do anything at all outside the "mainstream" of modern corporate computing, it's obviously because you're a sinister sociopath, lurking in the shadows until you can crash our national defense system. As I walked the aisles of the Las Vegas Convention Center, I wondered if the "Microcomputer Revolution" had taken 12 years to bring us back to minicomputers and a white-coated DP priesthood.

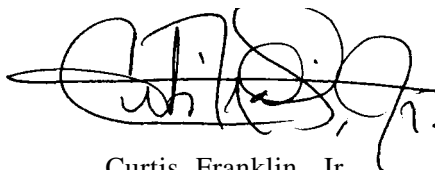
When I left the main hall of the show, and visited the smaller companies in the hotels, I finally saw evidence that the microcomputer revolution still flourishes. A number of companies showed me products that make putting your computer in touch with the real world easier and more effective than ever before. There were single-board computers and controllers in a few booths, and tools for hardware and software engineers and designers in a few more. Looking at the components, adapters, and tools made me realize that there is a lot of very high-quality work going on in the smaller companies within our industry. There seems to be something about a small company that leads engineers to concentrate on practical, cost-effective, real-world solutions to problems. It made me feel good when I found that these working problem-solvers are often Circuit Cellar INK readers. I finally decided that much of the truly revolutionary work going on today is happening in small companies, just like it did in the beginning. I like that, and I'm glad that Circuit Cellar INK is able to play a part in the ongoing computer revolution.

### Give Us an Earful

The response to our first reader survey was so helpful (Steve talks about the results of the survey in his column on the last page of this issue) that we decided to make it easier for you to tell us what you think of the magazine. Starting with this issue, there are three numbers printed at the end of each article. Please take a moment to decide which number matches your feelings about the article, go to the Reader Service Card in the back of the magazine, and circle the number. When you're through, tear out the card (it's postage paid) and drop it in the mail. The response we get to articles will have a direct impact on the type of articles we have in future issues. I'll say thanks in advance for helping us keep Circuit Cellar INK on the right track.

Beyond giving the editors feedback from you, the Reader Service Card that begins in this issue does two things: First, it makes it easier for you to get information on those products that interest you. Second, it gives our advertisers positive feedback on the investment they've made. Our readers are the reason we work so hard to make this magazine; our advertisers are the reason we can afford to make this magazine.

We're all looking forward to the new year. We're going to be growing and improving, but never losing sight of our mission to serve the people who design and build computer applications. From where I sit, it looks like a fun year ahead.



Curtis Franklin, Jr.  
*Editor-in-Chief*

EDITORIAL  
DIRECTOR/  
FOUNDER  
*Steve Ciarcia*

PUBLISHER  
*Daniel Rodriguez*

ASSOCIATE  
PUBLISHER  
John Hayes

EDITOR-in-CHIEF  
*Curtis Franklin, Jr.*

TECHNICAL  
EDITORS  
*Ken Davidson  
Jeff Bachiochi  
Edward Nisley*

CONTRIBUTING  
EDITOR  
*Thomas Cantrell*

CONSULTING  
EDITOR  
*Harv Weiner*

CIRCULATION  
COORDINATOR  
*Rose Mansella*

CIRCULATION  
CONSULTANT  
*Gregory Spitzfaden*

PRODUCTION  
MANAGER  
*Tricia Dziejdzinski*

BUSINESS  
MANAGER  
*Jeannette Walters*

STAFF  
RESEARCHERS

**Northeast**  
*Eric Albert  
William Curlew  
Richard Sawyer  
Robert Stek*

**Midwest**  
*John Elson  
Tim McDonough*  
**West Coast**  
*Frank Kuechmann  
Mark Voorhees*

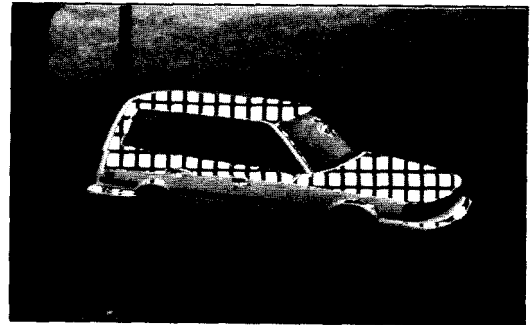
# Circuit CELLAR INK<sup>®</sup>

Ctrl

## FEATURES

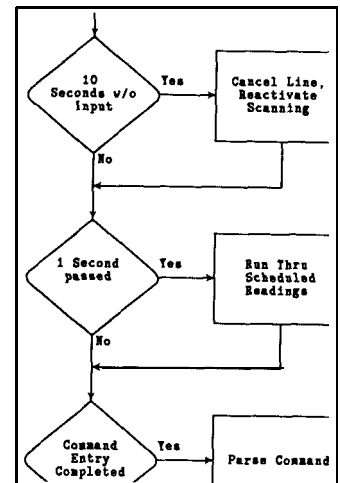
### 8 ImageWise/PC -- The Digitizing Continues -- Part 2 c 1 The Hardware by Ed Nisley

In the second of three parts, Ed Nisley shows the hardware details of this ISA bus gray scale digitizer, and contrasts the design with that of the original serial ImageWise. The starting point is the fundamental design decisions for an IBM/PC I/O bus board.



### 22 Build a Remote Analog Data Logger -- Part 2 The Software by R. W. Meister

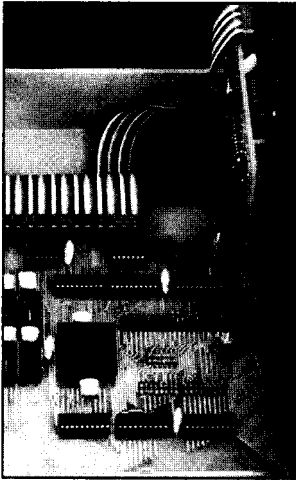
In the final installment of the article, we explain the C language software for the Motorola 6809 controller. Bob Meister uses descriptions and examples to cover program logic flow, interrupt handling, and coding for specific routines.



## DEPARTMENTS

Editor's ink The Revolution Continues by Curtis Franklin, Jr.	_____	1
Reader's Ink -- Letters to the Editor	_____	5
Visible Ink -- Letters to the CCINK Research Staff	_____	20
Ink Spot -- Guest Editorial A Call for Dedication by Ezra Shapiro	_____	34
From the Bench AC Power Line Transmission Conducted by Jeff Bachiochi	_____	42

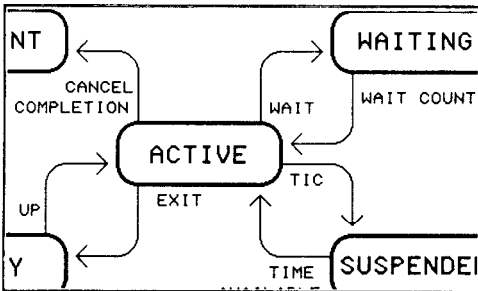
Circuit Cellar BBS - 24 Hrs. 300/1200/2400 bps, 8 bits, no parity, 1 stop bit, 203-871-1988



**36** The Home Satellite Weather Center -- Part 7  
*Finishing the Firmware for the 68000 Peripheral Processor*  
 by Mark Voorhees

As the 68000-based Peripheral Processor moves closer to completion, Mark Voorhees wraps up the controlling firmware for the system. This installment shows how to integrate Weather Facsimile (WEFAX) reception and interface the Heathkit ID-4001 and ID-5001 weather instruments to the Peripheral Processor.

**45** Writing A Real-Time Operating System -- Part 1  
*A Multitasking Event Scheduler for the HO64180*  
 by Jack Ganssle



In the first of two parts, Jack Ganssle explains the concepts behind a real-time operating system (RTOS) and shows how an RTOS differs from a general-purpose operating system. He then moves into the beginning of the actual code with the scheduling algorithm and central Task Control Block.

The schematics provided in Circuit Cellar INK are drawn using Schema from Omation Inc. All programs and schematics in Circuit Cellar INK have been carefully reviewed to ensure that their performance is in accordance with the specifications described, and programs are posted on the Circuit Cellar BBS for electronic transfer by subscribers.

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 the possible variation in the quality and condition of materials and workmanship of reader-assembled projects, Circuit Cellar INK disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in Circuit Cellar INK.

CIRCUIT CELLAR INK (ISSN 0896-8985) is published bimonthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (203-875-2751). Second-class postage paid at Vernon, CT and additional offices. One year (6 issues) charter subscription rate U.S.A. and possessions \$14.95, Canada \$17.95, all other countries \$26.95. All subscription orders payable in U.S. funds only, via international postal money order or check drawn on U.S. bank. Direct subscription orders to Circuit Cellar INK, Subscriptions, 12 Depot Sq., Peterborough, NH 03458-9909 or call (203) 875-2199. POSTMASTER: Please send address changes to Circuit Cellar INK, Circulation Dept., 12 Depot Square, Peterborough, NH 03458-9909.

Entire contents copyright 1988 by Circuit Cellar Incorporated. All rights reserved. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

Advertiser's Index	48
Firmware Furnace Real Numbers Number Crunching for the 8751 by Ed Nisley	52
ConnecTime -- Excerpts from the Circuit Cellar BBS Conducted by Ken Davidson	59
Steve's Own Ink First INK Reader Survey by Steve Ciarcia	64

Cover Illustration by Robert Tinney

## RE A D E R'S I N K

*Letters to the Editor*

I'd Like to See ...

For future editions, please consider a digital music mixer for the IBM PC/AT (including a MIDI interface), or a video-recorder data-backup device, or a video camera-to-IBM disk device.

John Lee  
Chicago, IL

*[For a video camera-to-disk device, I think that the ImageWise/PC, which continues in this issue, will be hard to beat. I haven't seen a project for any sort of data backup device here, but I would certainly be happy to look at a proposal from one of the Circuit Cellar INK readers. Finally, we're planning an issue on "Applications in the Arts" for later this year. If anyone has done work with MIDI, graphics, or other artistic applications, drop me a line to see about article possibilities. Ed.]*

I received issue #5 this week and thoroughly enjoyed it. I especially enjoyed the article on "10-MHz/8-bit Digitizing Board for the IBM PC." I would like to see more articles on video design including articles on frame grabbers, text overlay, and simple image processing. Thanks again for an enjoyable publication.

Steve Horacek  
Boulder, CO

*[Gee, it looks like we started ImageWise/PC just in time! We're working on bringing more video projects to Circuit Cellar INK, so hang in there. I think that Circuit Cellar INK in 1989 will have some articles that are right up your (video) alley. Ed.]*

Thanks for the great magazine. I am pleased with

the issues so far. I am glad that you are publishing technical articles of all types, not just strictly computer articles.

I hope that we'll see some articles written by Apple II enthusiasts. I'd like to see more information on the GIF graphics format mentioned in Circuit Cellar INK #3. I'm disappointed at the lack of Apple II software for the ImageWise digitizer. Perhaps GIF will be the answer.

Rolf Taylor  
North Salem, NY

*[OK Apple fans, here's your chance. If you've written ImageWise software for the Apple II, or if you're building applications around this venerable platform, let me know. Circuit Cellar INK tries to present applications based on many different computing platforms, and is dedicated to no particular architecture or operating system. Ed.]*

I have worked with computers for 20 years, have progressed from computer operator to programmer to systems analyst to computer scientist, and have been delighted with being able to make any computer "sing" for me. I never had any interest in electronics until I saw Steve Ciarcia's "Why Microcontrollers" in the August 1988 issue of BYTE. That did it for me. I have now gone hog-wild, researching the University of Houston technical library and every other source I can get my hands on which will expand my understanding and knowledge as pertains to all forms of ICs and peripherals. I have also undertaken several concurrent 8031-controlled projects which I thought up.

Will you be addressing any aspect of ASICs and PLDs in future issues, or should I just go ahead and study other sources to satisfy my interest in that area?

Allen R. Summers  
Pasadena, TX

[We certainly do have projects which use ASICs and PALS, as well as tutorials on how to design with them, in the works. One of the problems in this area is trying to offer projects that don't require a \$30,000 development system, and we're working in that direction. Stay tuned. Ed.]

I have enjoyed all the articles in Circuit Cellar INK. I am very interested in video processing, and in converting from one standard to another. I also like electromechanical projects, and have built such things as solenoid- and servo-operated gearboxes for R/C models.

Keep up the X-10 projects. I have 14 modules around my house operating lights, fans, TVs, and so on from a programmable timer, a maxi controller, and two mini controllers. I have a separate system for my ham shack made up of recovered units and their controls purchased at a ham fair for \$4.

Adrian M. Zeffert  
East Northport, NY

## An Important Theme

The nicest thing about Circuit Cellar INK is that I feel good reading it. It doesn't make me feel guilty about having forgotten the propagation theory back in '63, but still does give me the information about new and fun projects I want to dream about (even if I can't afford to build all of them). I am glad that you provide the necessary background (such as the stepper motor article) without patronizing. It reminds me of the days on the roof tuning our cubical quad, with the transmatch my father and I built from scratch (coils, chassis, and even the six-inch-long variable capacitor -- plates and all) feeling sure we knew all there was to know on antennae.

More important, though, is the theme I think I sense here. A willingness to share, the courage to do it yourself in a snotty-pants peer society that thinks itself so high and mighty, riding on big expensive systems, but can any of the members of the society even begin to calculate what brings the signal across three inches of PCB into the DRAM? HA! In Circuit Cellar INK I see, once again, a hope for the generation my kids will have to be members of.

My only regret now is that I won't be able to access the Circuit Cellar BBS. Are there any plans to get a Tymnet link? How about opening a conference on CompuServe? If you do get Tymnet, Telenet, or a CompuServe conference, please let us international readers know!

Steve Chandler  
Israel

[First, thanks for the kind words. One of the principle motivations behind a publication like Circuit Cellar INK is the feeling that you can help people improve their engineering and design skills; that you can help them be better doers, not just better shoppers.

We are constantly looking at ways to make the Circuit Cellar BBS accessible to more people, and we've talked about doing all the things you mention. The biggest limit right now is human resources. There's one overworked engineer/editor who already spends several hours a day just keeping up with the current Circuit Cellar BBS. If we expand the system's scope, it will take at least one full-time person to direct traffic. We're still looking and talking, though, and you can be sure that any changes will appear in the pages of Circuit Cellar INK before they show up on the board. Ed.]

Create Professional **Quality** Circuit Diagrams with

# MacSchematic

from **Thinking Tools**

MacSchematic is a library of over 800 Electrical and Electronics symbols for professional quality circuit diagrams on the Mac.

Symbols are in both PICT format for MacDraw, MacDraft, or other CAD/Draw programs and in MacDraw II libraries.

MacSchematic symbols are object oriented for superior printer, plotter and laser output. They can be rescaled without losing their high resolution.

MacSchematic symbols are designed to snap to grid so their external connections fall on grid points and lines connect to symbols "on the dot".

Includes symbols from ANSI Standard Y32.2:

- Analog Components
- Gates/Digital Devices
- Industrial Wiring
- Ladder Diagrams

Not Copy Protected  
All Macs

Symbols shown at 50% Reduction.

To Order  
Send Check, MO, or PO:  
Thinking Tools  
2411-M Linden Ave  
Baltimore, MD 21217

For Information  
or Visa/MC Order  
Call (301)-383-6490

MacSchematic: \$80  
Demo + Manual: \$10  
plus \$5 shipping/handling

Corrections

Issue #5, Sept/Oct '88 -- 10-MHz/8-bit Digitizing board for the IBM PC  
Page 30 -- U3 pin 4 should go to -5V through R8. Pm 7 should go to +5V through R7. Pin 8 is unused.

Issue #6, Nov/Dec '88 -- ImageWise/PC -- The Digitizing Continues  
Pages 38 & 41 -- Swap photos 1 and 3, leaving the captions in place. The caption for Photo 3 should refer back to Photo 1.

Circle No. 124 on Reader Service Card

# ImageWise/PC -- The Digitizing Continues

by Ed Nisley

Perhaps the fundamental truth of engineering is that you can't have everything. There are always conflicting requirements: speed, power, board space, design time, parts use, and complexity must all be balanced against each other. Throughout this article I will discuss the tradeoffs we made in the ImageWise/PC design so you can see why it works the way it does.

Because most of you are familiar with the original ImageWise design presented in the May and June '87 Ciarcia's Circuit Cellar articles in *BYTE*, I'll concentrate on what's new and different about the ImageWise/PC hardware. The starting point is the fundamental design decisions for an IBM PC I/O bus board.

## Where's the Buffer?

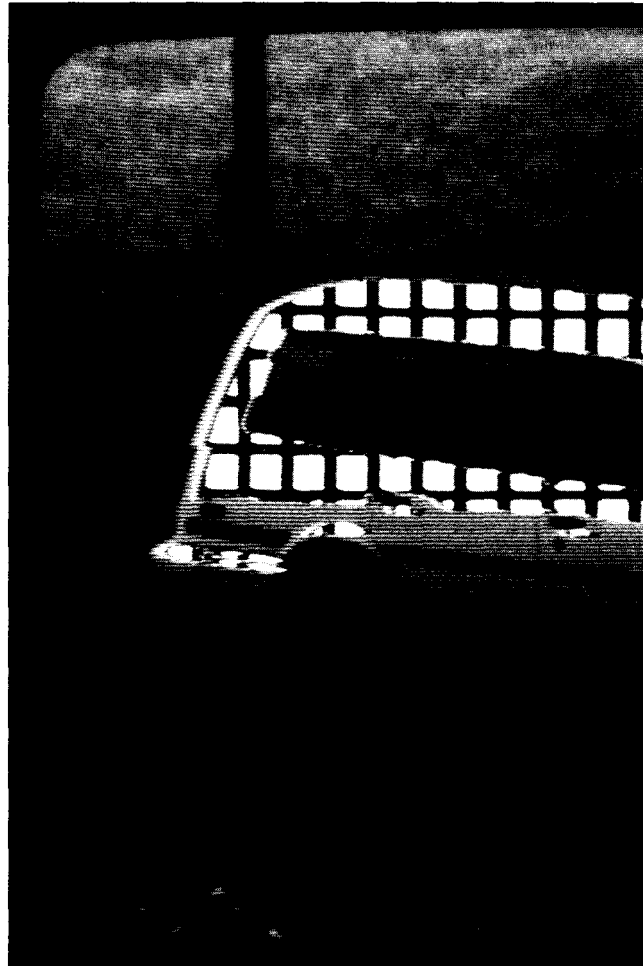
The heart of the ImageWise/PC circuitry is 64K bytes of static RAM, enough to hold one digital image. The video ADC (Analog-to-Digital Converter) translates each scan line of the incoming video signal into 256 separate pels (picture elements), so the buffer can hold up to 256 such lines, although one scan line is reserved for internal use. NTSC video has about 244 scan lines, which is the standard I will use throughout this article. PAL and SECAM video have about 280 scan lines, but only 255 can be held in the buffer. The

additional 25 lines normally fall within the monitor's overscan region, so aren't visible anyway.

The video ADC produces eight bits of data for each pel, resolving the image into 256 shades of gray. The video DAC translates this data back into an analog voltage resulting in a good reproduction of the original scene.

In order for the ImageWise/PC to be useful, the IBM PC must also have access to the buffer memory. The most obvious method is to assign 64K bytes of the PC address space to the ImageWise/PC video buffer. The memory maps in Figure 1 show why this simply won't work: while there are some 64K slots free in a bare-bones PC, a moderately well-equipped AT doesn't have any available memory addresses!

The AT's memory map does have some "holes" that are unused, so the ImageWise/PC buffer could be mapped into a hole in sections. For example, there may be 32K available in segment C000 hex next to the Disk BIOS ROMs. We decided that the added complexity



of the mapping hardware outweighed the advantages of direct access, particularly since the buffer's starting address would still have to be picked to match the peculiarities of each system.

We finally made a tradeoff that avoids memory mapping entirely! The video data can be moved through a single I/O port using I/O instructions instead of memory

## Part 2

# The Hardware

F000: 0000	ROM BIOS	ROM BIOS
E000: 0000	RESERVED	RESERVED
D000: 0000	FREE	EMS RAM
C000: 0000	FREE	DISK BIOS
B000: 0000	MDA BUFFER	EGA BUFFER
A000: 0000	FREE	EGA BUFFER
9000: 0000	RAM	RAM
8000: 0000	RAM	RAM
7000: 0000	RAM	RAM
6000: 0000	RAM	RAM
5000: 0000	RAM	RAM
4000: 0000	RAM	RAM
3000: 0000	RAM	RAM
2000: 0000	RAM	RAM
1000: 0000	RAM	RAM
0000: 0000	RAM	RAM

IBM PC MONOCHROME DISPLAY NO HARD DISK      IBM AT EGA DISPLAY HARD DISK LIMEMS RAW

**Figure 1 --**  
 A memory map for the PC/XT and PC/AT shows why using an I/O port was chosen for the ImageWise/PC as an alternative to memory mapping.

transfers. A counter on the ImageWise/PC board generates RAM addresses so the PC program can simply read or write bytes from the port.

I/O instructions are slower than memory transfers but the difference is relatively small. The key point is that even though the transfer takes somewhat longer, the ImageWise/PC board can be used in a "full-up" PC or AT with no problems.

### Access Control

The ImageWise/PC board uses an Intel 8031 microcontroller to handle many logic functions that would ordinarily require a lot of "glue" chips. While we could use an ASIC (Application-Specific Integrated Circuit) chip to merge these functions into a custom design, the microcontroller offers far more flexibility to handle require-

ments that crop up after the board is designed and manufactured.

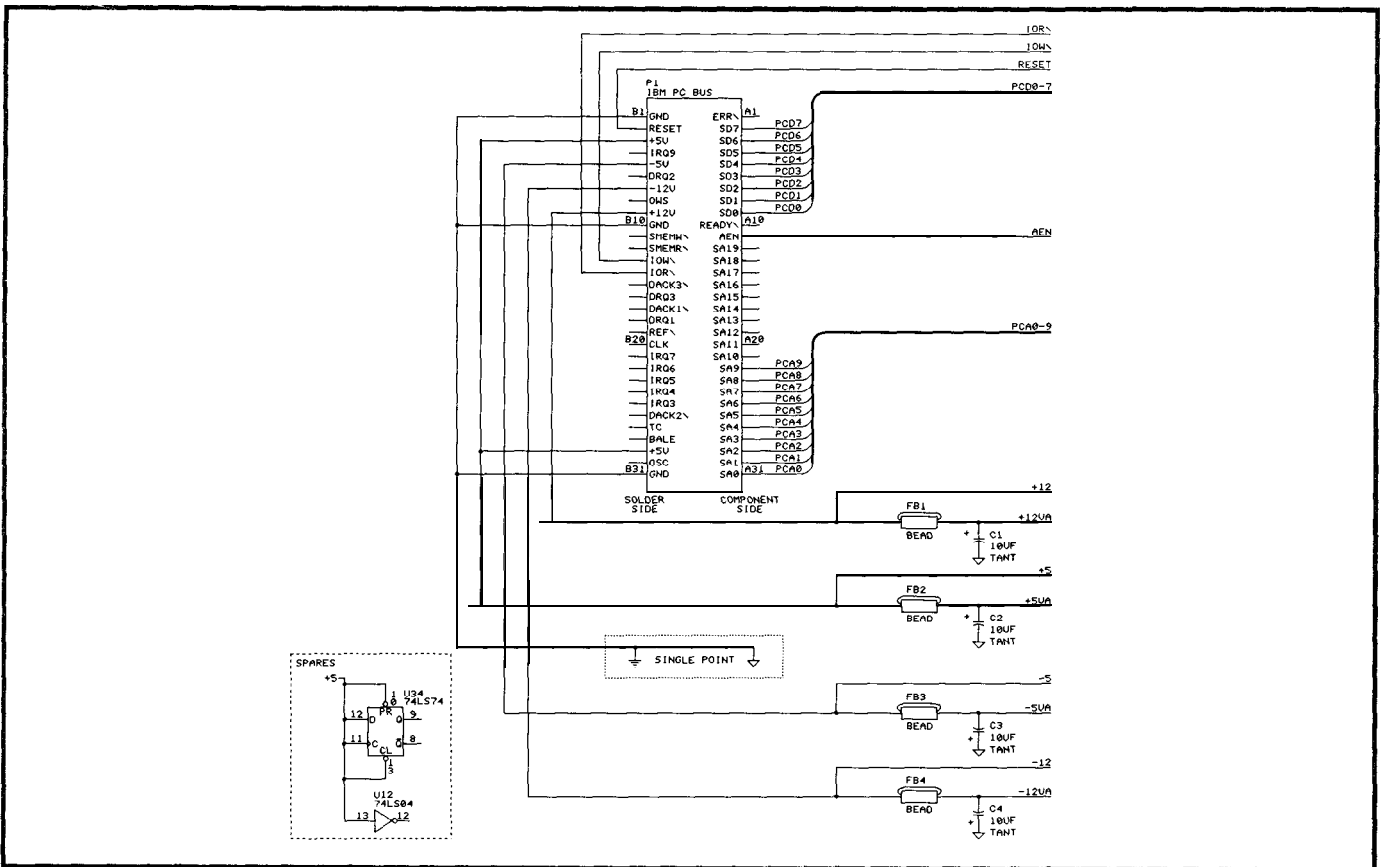
Because the 8031 can also read and write the video buffer, there are six ways to get data into or out of that RAM! Ideally we would like to allow simultaneous read/write access by the ADC, DAC, PC, and 8031. That way the software doesn't have to take conflicts into account. But there are some thorny problems along the way ...

Each scan line on the screen presents 256 bytes of data in about 51 microseconds, or one byte every 200 nanoseconds. The ADC will produce one byte and the DAC will consume one byte at that rate throughout the visible part of the image, although the ADC is active only when acquiring an image. Neither access can be delayed in the least because there is no way to "pause" the analog video signal. Therefore the ADC and DAC must have uninterrupted access to the Image RAM.

When the ADC or DAC are accessing the buffer, all other uses must be squeezed in between successive samples. Dividing the

*The ImageWise/PC can combine analog and digitized images as either digital over analog, or analog over digital image screens.*





**Figure 2 -- IBM PC bus connector and power supply filtering. Great emphasis was placed on power supply noise elimination because of the analog circuitry on the board.**

available time equally allows only 100 per access and allowing time for register and buffer delays means that the Image RAM must cycle in about 75 ns. While using a 64K buffer of 75-ns RAM is possible, the cost of the buffer and the additional circuitry gets out of hand quite rapidly.

We decided the best solution was also the simplest: allow only one device to access the Image RAM at any one time. The ADC and DAC have unrestricted use during active video, and the PC and 8031 must disable the video entirely or squeeze accesses into the blanking intervals to prevent collisions.

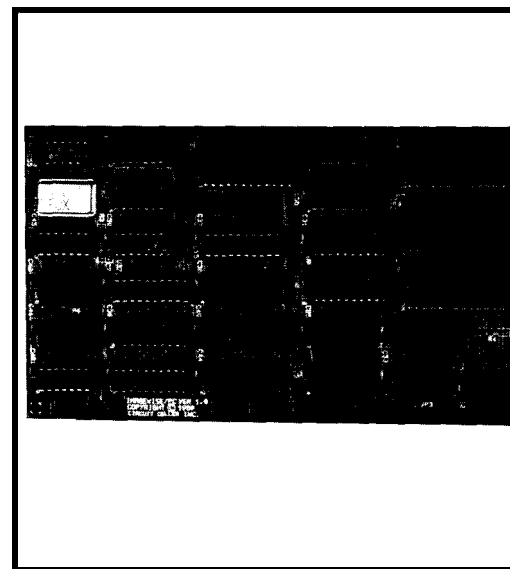
Now that you understand why the ImageWise/PC uses I/O-mapped accesses, it's time to get into the details. Figures 2 through 8 show the ImageWise/PC logic. Each schematic concentrates on the

logic for a particular section of the design.

### PC Interface Logic

From the PC's point of view, all access to the ImageWise/PC circuitry occurs through seven I/O ports. Two jumpers on the ImageWise/PC board set the base address for these ports, which is normally 110 hex. Figure 9 shows the default port assignments.

Figure 3 shows the port decoding logic and the registers that transfer data between the PC and the 8031 firmware. Although the board uses seven I/O addresses, it occupies 16 bus addresses because the decodes are not complete. For example, address bit 3 does not enter into the decoding logic so the board will respond to accesses at 110 or 118 hex with the same action.



The ports fall into three classes: hardware control, RAM access, and firmware interface. Because I started out by describing the reasoning behind Image RAM accesses, I'll begin with the RAM

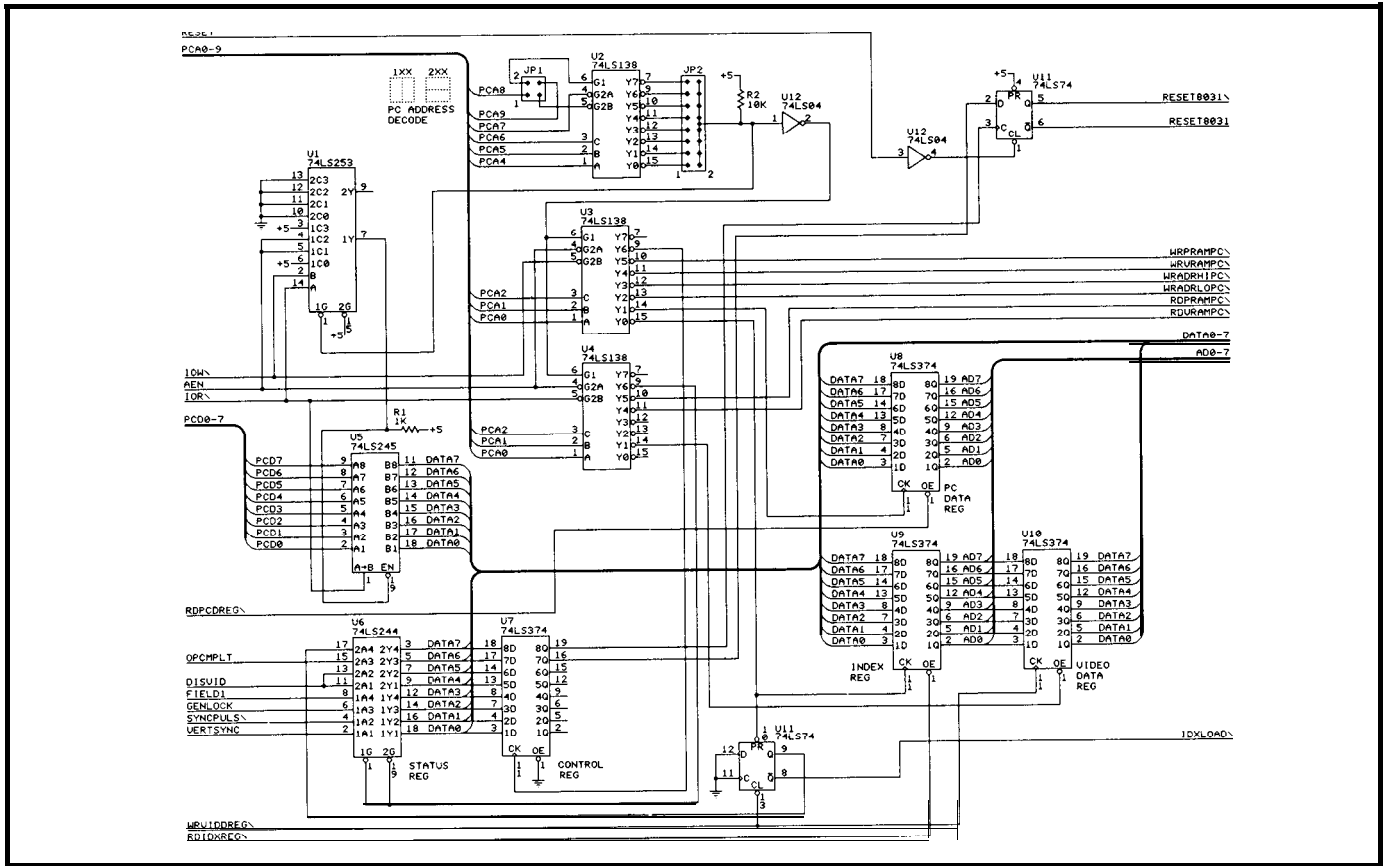
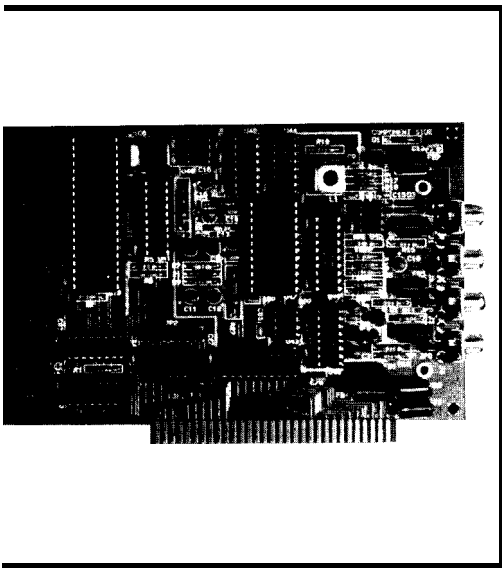


Figure 3 -- IBM PC bus interface circuitry. The board can be jumper-selectively addressed at one of 16 base port addresses.



The entire ImageWise/PC board fits into a single full-length 8-bit ISA (PC-Bus) slot.

control circuitry.

All image data is transferred through the VRAM port. Each access uses the current value of the PC Address Counter (U27 and U28), which is loaded by writes at ports ADDRHI and ADDRLO. Each VRAM access increments the address counter, so reading the full buffer requires a single address followed by 244 x 256 reads.

The port at CTL\_STAT contains all of the hardware control and status bits, which are itemized in Figure 10. Writes to this port set the controls, while port reads return the current status bits. I will describe the reset functions in more detail when I explain the Program RAM.

#### Firmware Interface

The INDEX and DATAREG ports provide a communication link

between the PC program and the firmware running on the 8031 processor. The board has great flexibility because nearly all of the board's functions can be adjusted on the fly. I will describe the firmware in more detail in the next article, but the hardware merits separate discussion.

There are several dozen variables controlling everything from the duration of horizontal sync pulses to the overall video signal level. These variables are grouped into the firmware registers shown in Figure 11.

Rather than provide a separate I/O address for each register, the ImageWise/PC firmware uses the INDEX port to specify which register will appear at the DATAREG port. Changing the contents of a firmware register requires writing the new value to DATAREG and writing the regis-

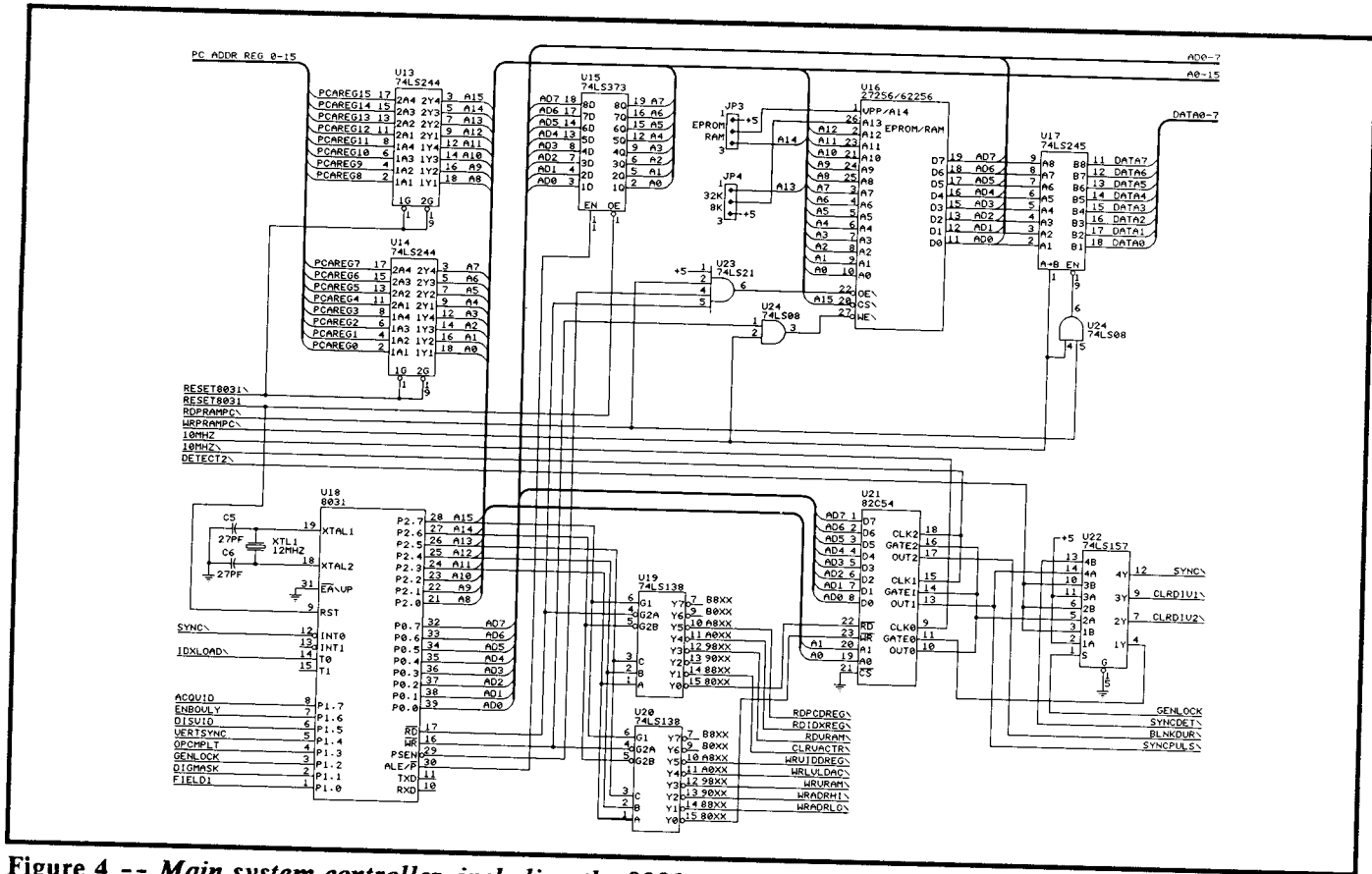


Figure 4 -- Main system controller, including the 8031 processor, control program storage, and programmable interval timer.

ter number to INDEX. Similarly, reading the current value is accomplished by writing the register number to INDEX and reading DATAREG.

### Program RAM

There is one port I have not mentioned yet: PRAM. This port gives access to the Program RAM holding the 8031 firmware program. In ordinary use, the PC will not have to read or write anything from this port because most Image-Wise/PC boards store their programs in EPROM.

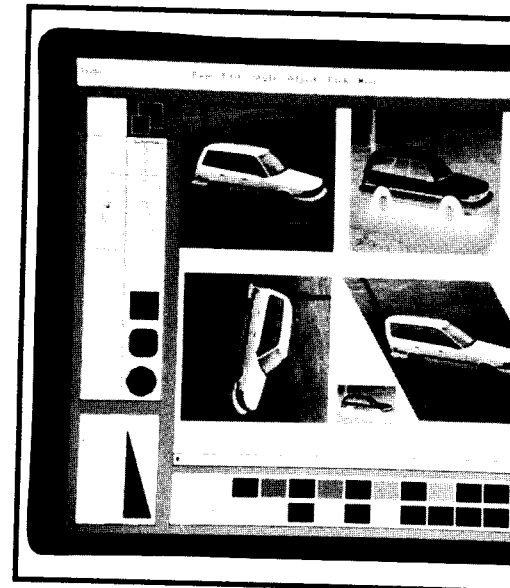
There are actually two DATAREG registers. U8 holds the value written by the PC program and U10 latches the corresponding value from the 8031 firmware. Each is readable by the other processor.

The 8031 monitors the output of U11B to discover when the PC writes an INDEX value. The U11B flip-flop is cleared when the 8031 writes a value into U10, which is ordinarily the last step in the firmware's response to the INDEX value.

The firmware presets the values of all the registers after a hardware reset so the PC doesn't have to load anything unless the defaults are not correct.

Close scrutiny of Figure 4, though, shows that the circuitry can handle static RAM chips or EPROMs, with either 8K- or 32K-byte capacity. Bus buffer U17 allows the PC to read and write the Program RAM much as it does the Image RAM. U17 is omitted from boards using an EPROM because there is no need to access the EPROM.

Because the 8031 must be reset during program loading, the Image RAM will be inactive. Rather than



include a separate Program RAM address counter, the PC Address Counter also provides the address for the Image RAM through multiplexers U13 and U14. These chips are omitted from EPROM-only

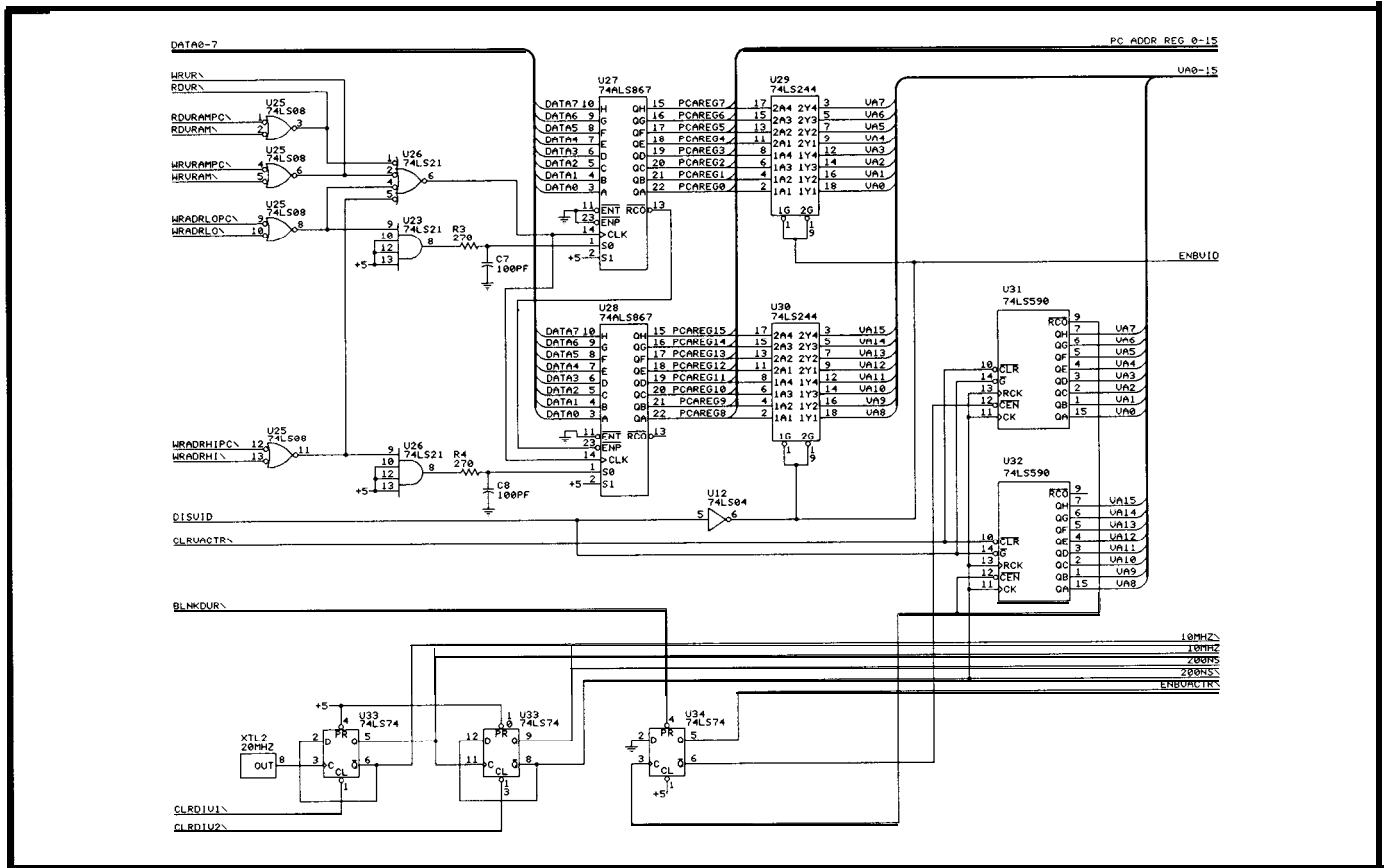
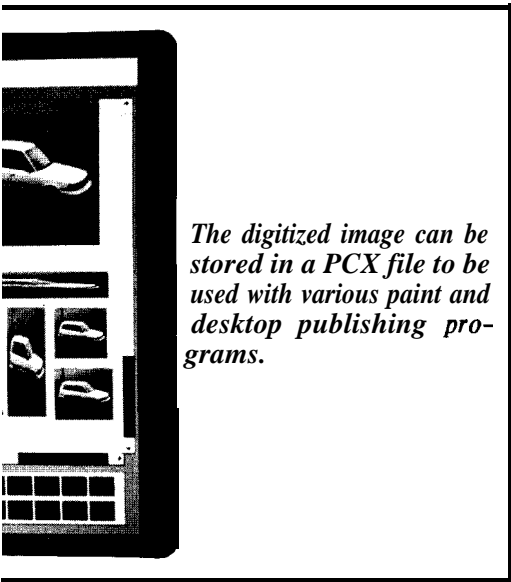


Figure 5 -- Video RAM addressing section plus master clock generation. Synchronous counters are used to simplify the task of addressing memory.



The digitized image can be stored in a PCX file to be used with various paint and desktop publishing programs.

boards.

Once the starting address is loaded into ADDRHI and ADDRLO, reads and writes to the PRAM port access the Program RAM. Unlike VRAM accesses,

though, these accesses do not increment the PC Address Counter. After reading or writing the PRAM port, the program must access VRAM to step to the next address.

RAM storage does introduce one complication, though: the ImageWise/PC cannot begin operation until the Program RAM contains a valid program. The solution to this is to provide a reset function controlled by an I/O port as shown in Figure 3 instead of using the I/O bus RESET signal.

The output of U1 IA is connected to the 8031 Reset line so that the processor is disabled whenever the flip-flop is cleared. A bus RESET signal clears the flip-flop, ensuring that the 8031 is halted when you turn on the power. Bits 6 and 7 in the CTL\_STAT port control the state of the latch; bit 6 sets or clears it, while a O-to-1 transition on bit 7 clocks bit 6 into

the latch.

### Analog Video Input

The analog video input circuitry shown in Figure 7 has several improvements over the stand-alone ImageWise design, although you're certain to recognize the similarities.

The circuitry must clamp the incoming sync tips to ground level so that the video signal levels are at a known voltage. The new design has better temperature compensation and clamps the tips closer to ground level.

The older design used a simple RC filter to eliminate the subcarrier from color signals. Although we intended that board for use with monochrome cameras, I have to admit we should have done better. The new chroma trap was suggested by a caller on our BBS; it works just

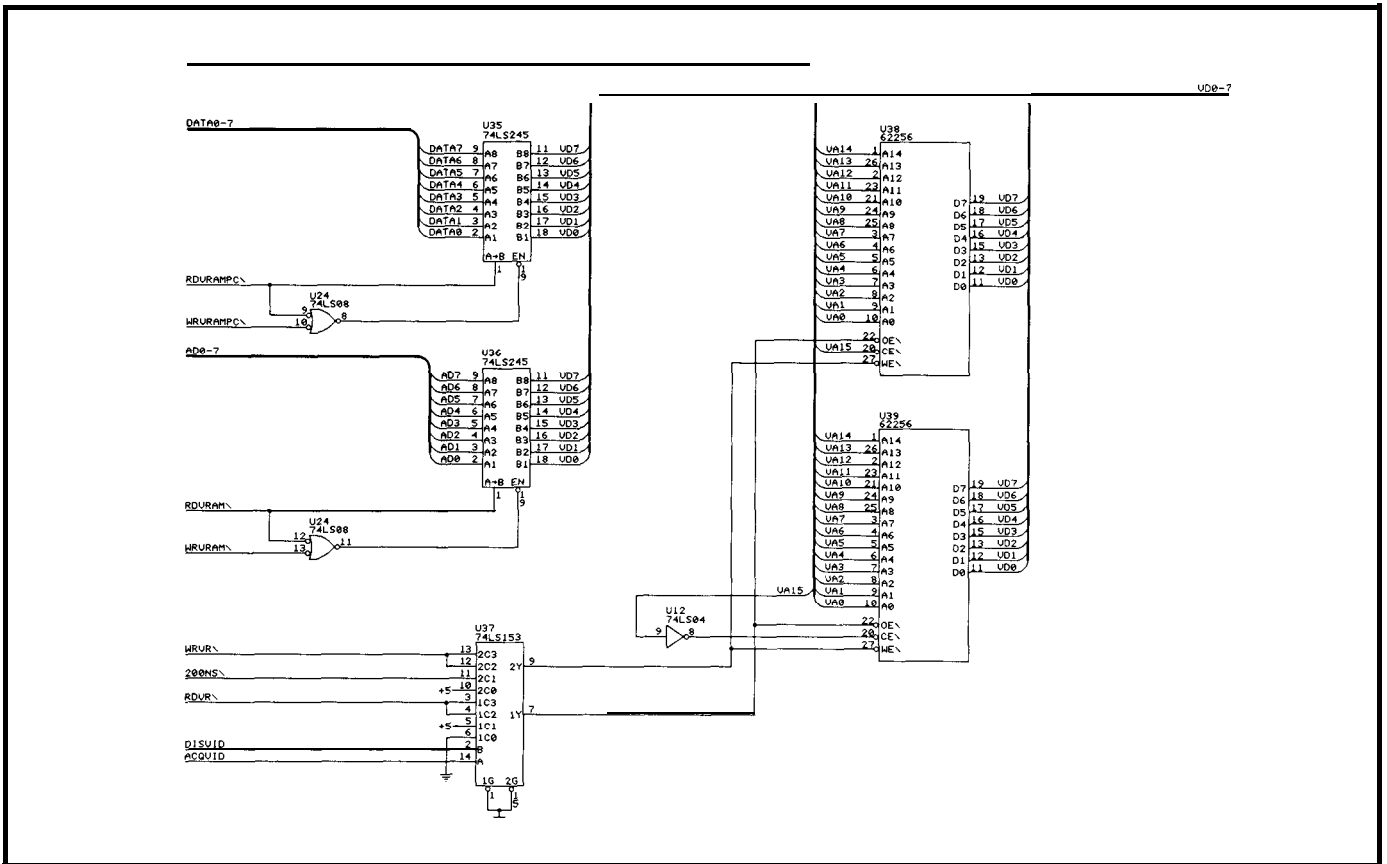


Figure 6 -- Video RAM and data buffers. One 256 x 244 picture fits nicely in the 64K of RAM provided on the board.

fine and we're glad to adopt it. Incidentally, the value of the inductor (L1) is different in the PAL and SECAM boards because the chroma signals use different frequencies.

The ImageWise/PC uses the CA3318 8-bit flash ADC (U47) to capture video signals. This IC is the "big brother" of the CA3306 used in the stand-alone serial ImageWise transmitter, but the circuitry required to drive it is significantly different.

Flash ADCs require one analog comparator for each voltage level, so the CA3318 has 256 comparators. The CA3306 needed only 64, so you can imagine the increased chip complexity! Each comparator decides whether the input signal exceeds a reference voltage set by a resistive divider driven from the Vref+ and Vref- inputs. A logic network encodes those 256 binary

signals into just eight output bits.

The increased number of resistors in the reference divider requires a higher voltage to provide sufficient reference current. Unfortunately, that means the analog signal must be higher as well, so we added an LF356 op amp (U46) to provide about 5 volts of video to the CA3318 input.

The reference voltages define the analog signal levels that will be converted into the 00 and FF (hex) digital codes. We used two outputs of the AD7226 quad DAC (U41) to set these voltages so you can match different camera levels without removing the PC's cover. The AD7226 doesn't have enough output drive capability to handle the CA3318 reference inputs, so Q4 and Q5 buffer the voltages.

An LM311 (U45) compares the video signal against a fixed reference to produce a digital signal

whenever a sync pulse occurs. As in the older design, the 8031 determines the pulse duration to find vertical syncs. Several people have asked where we hid the sync detection circuitry; there simply isn't any!

#### Analog Overlays

Although the ImageWise/PC uses the same TML 1852 video DAC (U40) to convert the digital values into analog waveforms, the surrounding circuitry as shown in Figure 8 is completely different! Because the same board has both the input and output video signals, adding some circuitry to overlay the two was both irresistible and fairly simple.

Although the term "overlay" implies that one image is placed atop the other, the actual hardware is a pair of switches (U42) with

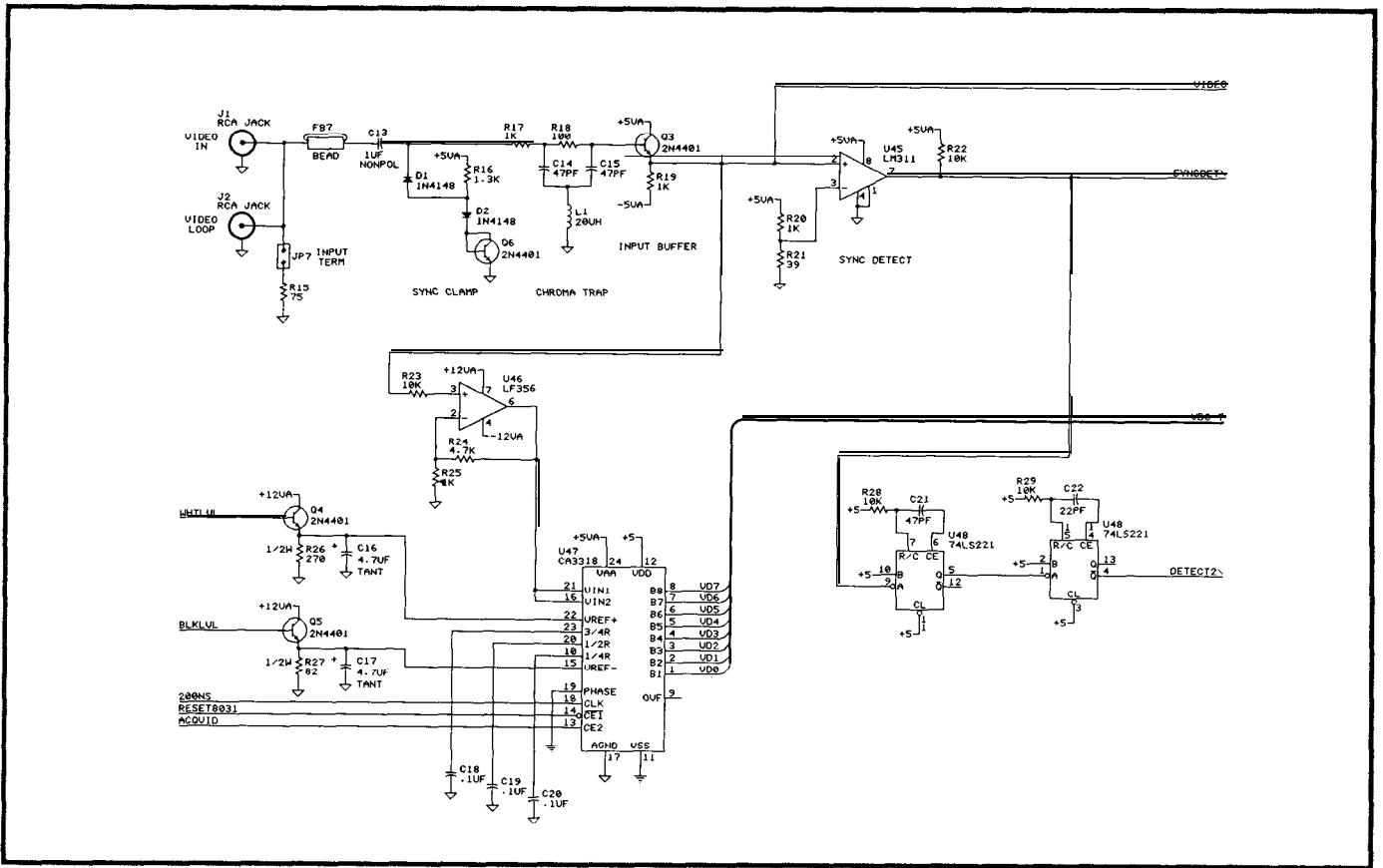


Figure 7 -- Video input circuitry. White and black reference level inputs to the ADC are programmable via the DAC found in Figure 8.

their outputs connected together. One switch is connected to the live analog video input, while the other is connected to the stored digital image. Each switch can transmit or block its video input; only one switch can be turned on at any time.

The analog switches are controlled by a second LM311 (U43) which compares a video signal against a reference voltage supplied by an output of the AD7226 quad DAC. Whenever the video exceeds the reference, the switches send the stored digital image to the output buffer. Conversely, when the video drops below the reference, you'll see the analog input.

We used the other two analog switches in U42 to select the overlay comparator's video signal input. This provides the flexibility to compare the level of either video signal; there are useful applications for both. Further, because the

overlay reference is set by a DAC, the switching level can be set anywhere from black to white.

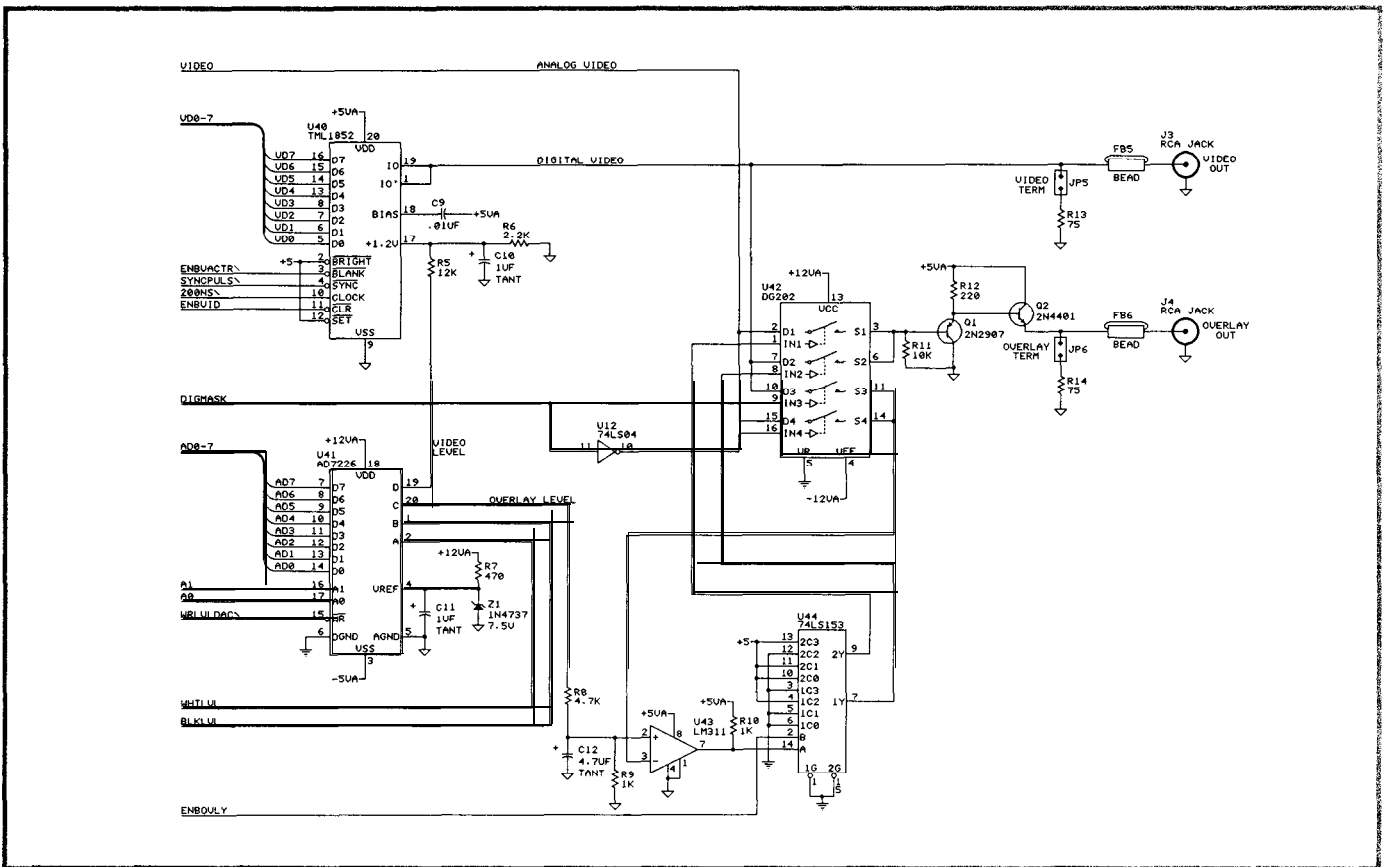
Of course, the TML1852 video DAC output also goes to a separate video jack so you can monitor the digital image directly even while an overlay is in effect. By setting the overlay level to a very low value you can see the entire live and digital images on two monitors at the same time, which simplifies camera focusing a lot!

The fourth AD7226 DAC output controls the amplitude of the TML1852 DAC, so you can adjust the output brightness directly. This is particularly useful because the signal level depends on the number of terminations on the DAC output. You can also fine-tune the level to match the analog input level so overlaid images are comparably bright.

## Genlock Sync

Creating overlays requires that the output sync pulses occur at the same time as the input pulses so that the two images are stable with respect to each other. TV production studios have a single sync generator that produces master timings for the entire installation; all of the video gear is "genlocked" to this single generator. The ImageWise/PC has a somewhat more relaxed definition of genlocking that is sufficient for our overlays.

Because both the input signal and the ImageWise/PC use crystal oscillators for their basic timing you might think that they are stable enough to get away with no locking at all. Even cheap oscillators have frequencies accurate to about 100 parts per million, which ought to be close enough. Unfortunately, that's just not so.



**Figure 8 -- Video output circuitry and level DAC. Note the solid-state switch used to provide overlay capabilities for the output.**

The basic line timing is 63.5 microseconds, so an error of 100 parts per million is 6.35 ns. After one frame of 525 lines the accumulated error is 3.33  $\mu$ s, which means that the scan lines would be displaced about 5%. After one second, the error is nearly 100  $\mu$ s, or one-and-a-half lines! Obviously we have to do much, much better than 100 ppm to have stable sync.

True genlocking requires an analog phase-locked loop that tunes the internal clock frequency to exactly match the external sync input. The design of this PLL is extremely critical for color TV signals because it must precisely match the phase of both color subcarriers. Because the subcarrier phase determines the colors you see on the screen, any error is immediately and painfully obvious.

The ImageWise/PC sync timings are created by an Intel 82C54

Programmable Interval Timer (U21) dividing a 10-MHz clock by the appropriate values to produce regular horizontal sync pulse intervals and widths, as well as horizontal blanking durations. U33A creates the 10-MHz clock from the 20-MHz output of XTAL2, a crystal oscillator.

The 10-MHz signal is further divided by U33B to get the basic 5-MHz pel clock used by the Image RAM circuitry. The hardware resets U33B at the start of each scan line, because otherwise the phase would flip on successive lines and cause obvious jitter. Think about it. . . the lines are 63.5 microseconds long and 5 MHz is 0.2 microseconds.

The 20-MHz oscillator is free-running and isn't synchronized to anything at all. When the ImageWise/PC is not genlocked, the 10-MHz divider is also free-running,

because there is no reason to reset it. The 5-MHz divider is reset on every scan line and all is right with the world.

When genlocking is in effect, U22 switches several signals that control the dividers and 82C54. The external sync signal now determines the horizontal line time, so 82C54-2 Timer 0 is programmed to create a short pulse immediately after the start of the external sync. The 10-MHz and 5-MHz dividers are both reset by pulses from U48 so that their phases have a known relationship to the input sync. Finally, the 8031 is interrupted by external sync instead of the 82C54 output so that it can track the external video.

Resetting the 10-MHz divider has an interesting side effect because it is the 82C54 PIT clock input. When the clock stops, all timings are suspended! The firm-

INDEX	110	index register and register write flag
DATAREG	111	data I/O register
ADDRLO	112	address register low byte
ADDRHI	113	address register high byte
VRAM	114	video RAM data I/O
PRAM	115	program RAM data I/O
CTL_STAT	116	control output/status input

Addresses are in hex.  
Default jumper settings are assumed.

**Figure 9 -- All access to the ImageWise/PC takes place through seven I/O ports. The default is for port 110 to be the beginning port for the ImageWise/PC.**

Bit	Writing	Reading
0	unused	+Index Loaded
1	unused	Operation Complete
2	unused	unused
3	unused	unused
4	unused	+Field 1
5	unused	+Genlocked
6	+Enable 8031	-Sync Pulse Active
7	+Strobe Enable Latch	+Vertical Retrace

**Figure 10 -- All of the hardware control and status bits for the ImageWise/PC are routed through the CTL\_STAT port at default address 116.**

Registers 00 through 0F contain control and status bits, set overall operating conditions, and handle other miscellany.

00	Program control bits
01	Program status bits (read only)
02	Setup bits
03-07	reserved
08	ADC white level
09	ADC black level
0A	Overlay switching level
0B	Digital video Output level
0c	Genlock retry limit
0D	reserved
0E	Firmware version (major and minor hexits)
0F	Firmware version (two's complement of Reg 0E)

Registers 10 through 1F control internal sync generation.

10-11	Horizontal line period
12-13	Horizontal sync duration
14-15	Horizontal blanking duration
16-17	Equalizing pulse duration
18-19	Vertical sync duration
1A	Number of active video lines
1B	Number of equalizing syncs
1C	Number of vertical syncs
1D	Number of syncs during vertical blanking
1E	reserved
1F	reserved

Registers 20 through 2F control external sync generation, which is used during genlock operation.

20-21	Horizontal sync delay
22-23	Horizontal sync duration
24-25	Horizontal blanking duration
26-27	Equalizing pulse duration
28-29	Vertical sync duration
2A	Number of active video lines
2B	Number of equalizing syncs
2c	Number of vertical syncs
2D	Number of syncs during vertical blanking
2E-2F	Camera horizontal blanking

**Figure 11 -- Variables stored in the firmware registers control most of the attributes of the ImageWise/PC. Quantifies occupying two registers (16-bit values) are stored with the least-significant byte in the lowest-numbered register. Time intervals are in units of 100 microseconds.**

ware compensates for the typical value of the reset pulse, so the resulting timings are still accurate.

Unlike a true analog PLL design, the ImageWise/PC genlock circuitry can only adjust the internal timings in units of 50 ns (half of the 10-MHz period). This is 1/4 of a pixel width and is generally not noticeable, but under some circumstances you may see a slight "crawl" on sharp vertical edges. Most of the cameras and monitors we've tested work just fine, so you probably will not encounter it.

And More to Come

Although we have gone through the hardware design, some parts of the ImageWise/PC can be understood only if you see how the firmware controls the gates. In the next article I will describe the firmware required to make the board work, as well as some interesting PC utility software.

IRS

- 201 Very Useful
- 202 Moderately Useful
- 203 Not Useful

The following is available from:

CCI  
4 Park Street, Suite 20  
Vernon, CT 06066

For information and orders:

Tel: (203) 875-2751

FAX: (203) 872-2204

**Item 1: ImageWise/PC PC board experimenter's kit**  
Comes with 4-layer PC board, and assorted key components including 27C256 EPROM, TML1852 D/A, AD7226 D/A, CA331 8 A/D, RCA jacks, ferrite beads, user's manual, and ImageWise/PC utilities on IBM PC format diskette.

Order IWPC-EXP ..... \$399.00

**Item 2: ZIP image processing software for ImageWise/PC from Hogware.**

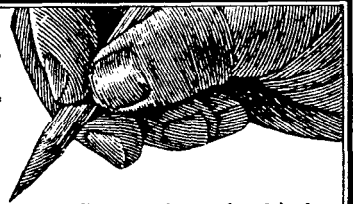
Or&r IWPC-ZIP ..... \$79.00

A procurement list for all ImageWise/PC board components is posted on the Circuit Cellar BBS. Full kits are not currently available. ImageWise/PC is available assembled and tested.

All payments should be made in U.S. dollars by check, money order, MasterCard or Visa. Shipping and handling: surface delivery add \$6.60 for U.S. Call for Canada and air freight delivery elsewhere.



# IT'S NOT TOO LATE !!



You have until May 1, 1989 to enter!

We want to find out just what kind

of applications designers the readers of Circuit Cellar INK are. To find the answer, we're sponsoring the first Circuit Cellar Design Contest. This is your chance to win the acclaim of your colleagues, the admiration of Circuit Cellar INK readers everywhere, and some pretty nifty prizes.

We've tried to make this a simple contest. The emphasis is on embedded control applications, and you can use any commercially available controller chip (8052, 8096, 6811, 8742, etc.) in your design. Our team of judges will be looking for utility, creativity, professionalism, and elegance in the designs. Within these easy guidelines, the choices are yours: make it prosaic or outlandish, simple or complex, the choice is yours!

**Prizes!** Of course there are prizes! **First** prize is \$500, **Second** prize is \$200, and **Third** prize is \$100. In addition, we'll award as many Honorable Mention prizes (\$50 and a 1-year subscription to Circuit Cellar INK) as are deserved. On top of all these prizes, winning entries may be the subject of full Circuit Cellar INK articles.

The winners will be announced in the July/August 1989 issue of Circuit Cellar INK. To enter, you must have an official entry form. To receive an official entry form, send a SASE to:

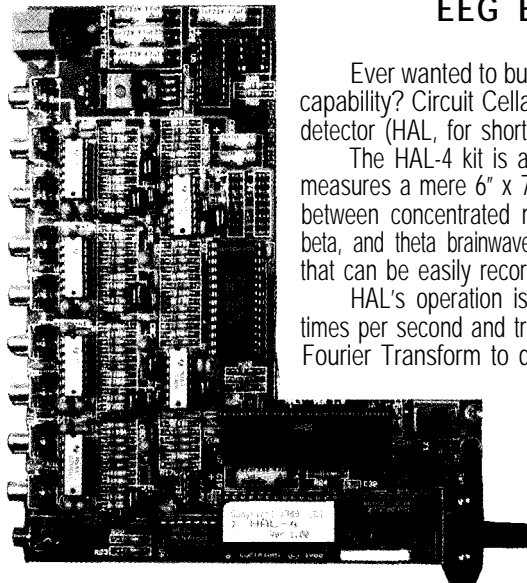
**Circuit Cellar INK  
Design Contest  
P.O. Box 772  
Vernon, CT 06066**

Build Steve Ciarcia's

## HAL=4

### EEG Biofeedback Brainwave Analyzer

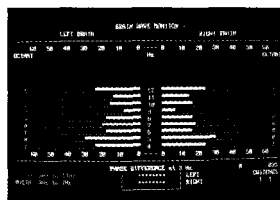
**FREE  
OFFER**



Ever wanted to build a brainwave analyzer; one that wasn't a toy; one with **graphice display** capability? Circuit Cellar is proud to introduce Steve Ciarcia's new Hemispheric Activation Level detector (HAL, for short).

The HAL-4 kit is a complete battery-operated **4-channel** electroencephalograph (EEG) which measures a mere 6" x 7". HAL is sensitive enough to even distinguish different conscious **states**—between concentrated mental activity and pleasant daydreaming. HAL gathers all relevant alpha, beta, and theta brainwave **signals within** the range of 4-20 Hz and presents it in a serial digitized format that can be easily recorded or analyzed.

HAL's operation is straightforward. HAL samples four channels of analog brainwave data 64 times per second and transmits this digitized data serially to a PC at 4800 bps. There, using a Fast Fourier Transform to determine frequency, amplitude, and phase components, the results are graphically displayed in real time for each side of the brain.



- Stand-alone-not bus dependent
- PC demo display software included
- RS-232 serial output
- \* Safe battery operation\*
- \* Complete software source code available

**HAL-4-KIT.....\$179.00**  
(plus S&H)

**\*\* WARNING:**

The Circuit Cellar Hemispheric Activation Level detector is presented as an engineering example of the design techniques used in acquiring brain-wave signals. This Hemispheric Activation Level detector is not a medically approved device, no medical claims are made for this device, and it should not be used for medical diagnostic purposes. Furthermore, the safe use of HAL requires that the electrical power and communications isolation described in its design not be circumvented. HAL is designed to be battery operated only!

To order please call **CCI: (203) 875-2751**  
**TELEX: 643331**  
**FAX: (203) 872-2204**

**Circuit Cellar Inc.**  
4 Park St.- Suite 12  
Vernon, CT 06066

**SPECIAL:** Mention this ad when placing your order and receive a set of HAL disposable electrodes.....**FREE**

# VISIBLE INK *Answers; Clear and Simple*

---

## *Letters to the INK Research Staff*

### Field Data Collection

I suppose this is just one of the tons of letters you receive each day, so I'll be brief. The subject of my questions concerns field data collectors, known also as portable data terminals. Before replying, I want you to know that your answers to the following questions will be highly appreciated.

What is the state of the art in field data collectors? What are their virtues and drawbacks? Are they expensive? What are the costs for a given storage capacity and quality? Are costs expected to go down significantly in the near future?

I'm interested in this matter because I am finishing the design of a field data collecting system. Its main (not all) features are:

#### Collector:

- Size: 110 x 70 x 23 mm (4.33 x 2.75 x 0.9 in.) (about the size of a cassette box, but thicker)
- Weight: 10 oz.
- Storage: 48 Kb
- One- hand operation
- Waterproof, dustproof, shockproof
- Protected against electromagnetic and radio frequency interference
- Operating temperature: -25 to 160°F
- Altitude: up to 25,000 feet
- Price: Approx US\$300
- Ability to scan recorded data, to delete last entries, to label bad data, etc.

#### Interface:

- Size: 110 x 70 x 35 mm (4.33 x 2.75 x 1.37 in.)
- Weight: 25 oz.
- Data format: ASCII
- Data Transfer: Serial RS-232

- Data Rates: 300, 1200, 4800, 9600 bps
- Collector not needed to transmit data
- Price: Approx US\$300

Do you think there's any chance for a system like this to be sold reasonably in the market?

I'll be very grateful for your help in all these points, and any other that you might consider important.

**Roberto Garcia S.**  
Chile, South America

***There are several subdivisions of the data collector market, each specialized for a particular type of data. We'll go over what we know on the subject so you can decide if your devices fit.***

***The simplest data collection devices are intended for manual data entry at a location where a standard PC isn't practical. These are typically a keyboard of some sort, a display (usually an LCD with one to four lines and 16 to 80 characters), and an 8-bit microprocessor driving some battery-backed RAM. Usually there is a serial link used for uploading the collected data. This sounds like the type of device you are describing.***

***Next in line are collectors that have analog and digital inputs for direct connection to measuring devices. The controller will have some programming capability, generally a dialect of BASIC, to allow device setup and calibration. These are more expensive because they need precision analog circuitry and a more capable computer (perhaps with 16 bits or a faster clock).***

***Finally, there are complete data acquisition and control systems that can collect, analyze, and transmit data in real time. These are more like "real" computer systems and are usually installed in a fixed location.***

*They're hardened against environmental stress and may come in rack-mount versions for industrial applications.*

*Although there is some market for the "remote terminal" type of data collector, these devices are being squeezed by the low end of the laptop computer market. For about US\$700 you can now purchase a Toshiba 1000 laptop: a PC clone with a 25x80 LCD display, full keyboard, 720K floppy drive, serial and parallel ports, and 640K of RAM.*

*Now, admittedly, the Toshiba is much larger and heavier than your collector and it's not hardened against the full range of environmental hazards. On the other hand, it's only about US\$100 more than your device and is far more capable. In fact, what some folks do is write a dedicated program that handles full-screen data entry and editing and saves the result on diskette. The user doesn't even know that the computer can do anything else! For most applications, the Toshiba makes a lot of sense.*

*In short, it's quite likely that your collector has value for the market niche that needs small size, light weight, and relatively limited data entry capabilities. However, you will have to convince yourself that it can compete against the devices already on the market that are attempting to fill that niche, as well as survive against more capable units like laptops that will always have a significant cost advantage because of their volume production.*

*You must make a very careful market survey before proceeding. We can't help you with that because you need to collect specific information on all the devices currently available, decide what features the "next generation" will have, and how your device will be positioned against them.*

### Keeping Lightning off the Lines

I have just completed a power strip built into an Apple II power supply enclosure with an EMI filter and three MOVs. The question is, can I be fully assured that, in case I forget to turn this thing off and it gets hit by lightning, the EMI line filter will get the kick since it is wired closest to the line cord?

Next, what kind of similar protection can I use for modems? I believe there are commercial gadgets for this, but what about a DIY version?

James Lek  
Singapore

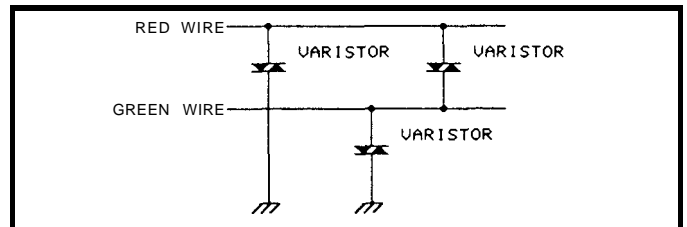
*The power strip you describe should, if properly constructed, function well in normal use. However,*

*there is nothing that will offer complete protection for equipment (including the outlet box itself) from a direct or nearby lightning strike.*

*If frequent lightning strikes in your vicinity pose a problem, the best remedy is probably an array of fast-acting lightning arrestor equipment installed where the power line enters your building. Such protection is expensive and therefore reasonable only in unusual circumstances.*

*We assume that, in your desire to protect a modem, you want surge suppression on the phone lines rather than, say, on an RS-232 hook-up. With all the different technical specifications for telephone systems in various parts of the world, it's difficult to give an exact MOV circuit. However, we can provide a diagram showing a common method for phone line protection here in the U.S.*

*As the diagram below shows, there are commonly two phone wires, one with red insulation and one with green insulation. One MOV is installed across these wires: another MOV is installed between each wire and earth ground. Since the peak ring voltages sometimes approach those of the AC line, 130-VRMS MOVs are used. With suitable adjustments to allow for conditions in your country, this type of circuit offers good protection at reasonable cost.*



### IRS

- 204 Very Useful
- 205 Moderately Useful
- 206 Not Useful

In Visible Ink, the Circuit Cellar Research Staff answers microcomputing questions from the readership. The representative questions are published each month as space permits. Send your inquiries to:

**INK Research Staff**  
c/o Circuit Cellar INK  
Box 772  
Vernon, CT 06066

All letters and photos become the property of CCINK and cannot be returned.

# Build a Remote Analog Data Logger

by R.W. Meister

## Part 2 *The Software*

**I**n the first part of this article (Circuit Cellar INK #6), I described the hardware for a remote data logging system. In this final portion of the article, I'll talk about the software that controls the hardware, why I wrote it the way I did, and how it works.

### Why C?

Although C is considered a high-level, third-generation language, it can reference specific addresses in memory and communicate with hardware at a level that approaches assembly language. I could have written this program in assembly language, which would have provided us faster execution time (in an environment that relies on delays inherent in the hardware), but no one else would have been able to maintain the program. In addition, assembly language would have restricted the program to one kind of processor. Anyone with a different CPU would have to know both M6809 assembly language and their own CPU's assembly language, making translation difficult if not impossible. By using a high-level language, I have attempted to remove the CPU dependency from the program. The limited amount of assembly language is fairly easy to implement on other processors and may not even be required in some implementations. Of course, my own familiarity and preference, as

well as the "challenge" of the project, had something to do with the choice.

The software was a major part of this project. Parts of the 1000+ lines of heavily commented C code are being reproduced so you can refer to the program while reading this text.

**[Editor's Note: Complete software for this article is available for downloading from the Circuit Cellar BBS or on Software On Disk #7. For downloading and ordering information, see page 62.]**

Most C programs, when compiled, rely on rather extensive libraries of support subroutines, many of which are large, to handle transcendental functions such as sine, cosine, and square root, and the usually necessary input and output operations to terminals and storage devices. Even a simple program that prints "hello" can produce an executable file as large as IOKB! This program was designed to be fully self-contained (i.e., not require any external library subroutines) and makes only minimal use of assembly language routines. It is very modular, with only one routine spanning more than one page of code in the listing. Because of the lack of support subroutines from libraries, terminal input and output routines had to be written for the A/D program. There are no storage devices to deal with, and limited formatted output.

### Hardware-Specific Programming

I wrote two interrupt service routines to do very small amounts of work in the shortest time possible, essentially setting flags, storing characters, or counting events. One handles the real-time clock, the other handles characters coming in from the device connected to the serial interface. The M6809 CPU handles interrupts by saving all of its registers on a stack. It then transfers program control to the interrupt service routine which, when finished, returns control to the interrupted program using an instruction that restores all of the saved registers. These service routines had to be written in assembly language to properly interface to the M6809. These routines then call the C subroutines directly. There was also an anomaly in the C compiler that was used on the M6809 development system: it used subroutines to perform multiplication and division when required by the running program. Some other systems use in-line code or even single CPU instructions for these operations. These routines were disassembled and rewritten in assembly language as part of the source program, again to preclude the necessity of needing any support library functions. A small initialization routine was also written in assembly language to set the hardware stack pointer, enable interrupts, and call the main C function.

There are several interrupt inputs on the M6809. One, called the Non-Maskable Interrupt (NMI), is always recognized by the CPU and can never be ignored or masked. The others are maskable, meaning that the CPU can suspend their action under program control for some amount of time. The

A/D box uses NMI for the clock and the Interrupt Request (IRQ) line for characters from the ACIA.

The nonmaskable interrupt is generated by the 601-Hz signal from the data rate circuit. This event calls a C function that decrements a "tick" counter. When it reaches 0, it is reset to 601, and a

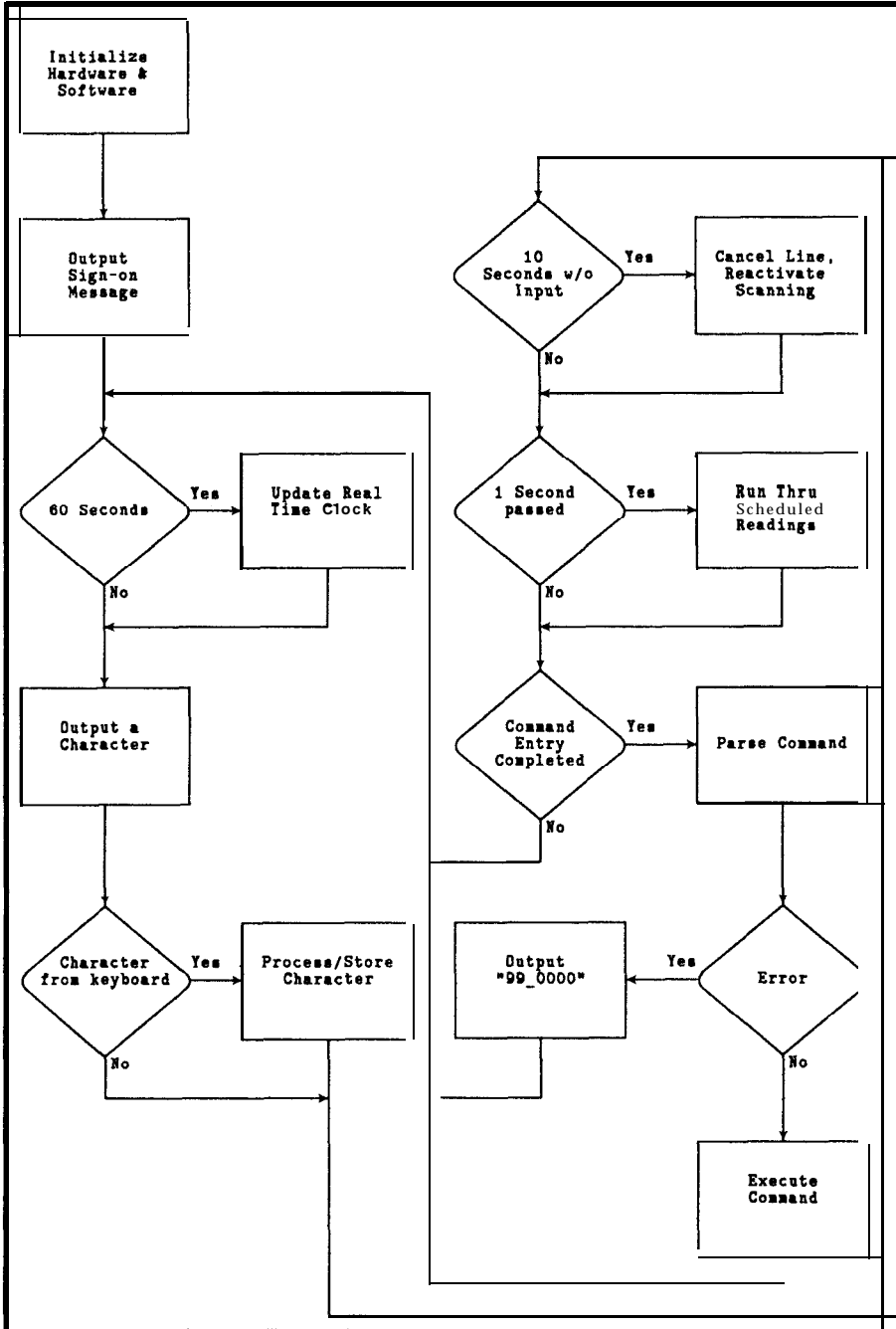


Figure 1 -- The flowchart for the data logging software shows that the program exists as a large loop that waits for data, and calls one of several subroutines when necessary. Modular programming and polling techniques mean that there are few situations when data is lost because of conflicts over CPU time.

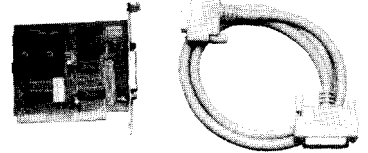
## JDR Microdevices

- 30 day money back guarantee
- 1 year warranty on all products
- Toll-free technical support

### New! Modular Programming System

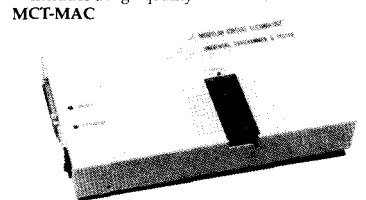
FROM MODULAR CIRCUIT TECHNOLOGY

This integrated system is ideal for developers--it easily expands as your needs grow! All the modules use a common host adaptor card so you need just one slot to program EPROMS, PROMS, PALS and more!



#### Host Adaptor Card \$29.95

- A universal interface for all the programming modules
- User-selectable programmable addresses prevents addressing conflicts
- Includes a high quality molded cable



#### Universal Module \$499.99

- Programs EPROMS, EEPROMS, PALS, bi-polar PROMS, 8748 & 8751 series devices.
- Programs 16V8&20V8 GALS (gallium arsenide) from LATTICE, NS, SGS
- Tests TTL, CMOS, Dynamic & Static RAMS
- Load disk, save disk, edit, blank check, program, auto read 'master, verify and compare
- Textool socket accents 3" to .6" wide IC's from 840 pins

#### EPROM Module \$119.95

- Programs 24-32 pin EPROMS, CMOS EPROMS and EEPROMS from 16K to 1024K
- HEX to OBJ converter
- Auto, blank check/program/verify
- VPP selectable 5, 12.5, 12.75, 13.21 & 25 volts
- Normal, intelligent, interactive, and quick pulse programming algorithm

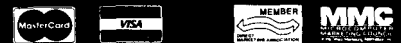
MCT-MEP  
 MCT-MEP-4 I-EPROM Programmer \$169.95  
 MCT-MEP-8 R-EPROM Programmer \$259.95  
 MCT-MEP-1616-EPROM Programmer \$199.95

#### PAL Module \$24 9.95

- Programs MMI, NS, TI 20 & TI 24 pin devices
- Blank check, program, auto, read master, verify and security fuse blow

MCT-MPL  
 PAL Programming development software \$99.95  
 MC6MPL-SOFT

Order toll free 800-538-5000



JDR MICRODEVICES, 110 KNOWLES DRIVE, LOS GATOS, CA 95030. LOCAL: (408) 866-6200 FAX (408) 378-8927  
 RETAIL STORE: 1256 S. GASCORN AVE. SAN JOSE, CA  
 HOURS: MON.-FRI. 9-7, SAT. 9-5, SUN. 12-4 (408) 947-8881

Terms: Minimum order \$10.00. For shipping and handling include \$3.90 or ground and \$4.50 air. Orders over 1 lb and foreign orders may require additional shipping charges--please contact the sales department for the amount. CA residents must include applicable sales tax. Prices are subject to change without notice. We are not responsible for typographical errors. We reserve the right to limit quantities and to substitute manufacturer. All merchandise subject to prior sales. A full copy of our terms is available upon request. Items pictured may only be representative.

COPYRIGHT 1989 JDR MICRODEVICES

## Need A Sophisticated Logic Analyzer But Cannot Afford One ?

### the **ID320**

Logic Analyzer Card  
for the IBM PC/XT/AT  
May be Your Answer

State of the art design brings  
you high-end performance at  
low-end price

#### Features:

- 32 channels 2K deep
- 25 MHz state analysis  
50 MHz timing analysis
- Variable Threshold  
Input Pods
- Single Slot Design
- Multi-level triggering
- Selective Data Capture
- Software Performance  
Analysis
- Disassemblers for  
popular 8/16 bit  
microprocessors
- Test Development  
Language supports ATE  
applications
- Friendly User Interface

From \$1395

Satisfaction Guaranteed  
or your money back

 **INNOTECH  
DESIGN INC.**

14640 Firestone Blvd., Ste. C  
La Mirada, CA 90638  
Tel: 714-521-5454

"seconds" counter is incremented. The CPU then returns to the part of the program that was being executed when it was interrupted. Later on, the program recognizes that the "seconds" have changed, and adjusts the real-time clock. In a similar manner, characters received by the ACIA generate an interrupt that calls a C function. This function resets some pointers if the input buffer is empty, deals with Ctrl-S (XOFF) and Ctrl-Q (XON) by setting or clearing a flag, and then stores the character in the input buffer. The M6809 logic disables (masks) any further interrupts from the ACIA while the interrupt service routine is executing. When it finishes, the CPU will acknowledge and deal with any other interrupts that have occurred while it was busy. In actuality, the 4800-bps maximum data rate gives the interrupt service routines all the time they need to correctly process interrupts.

#### Into the Main Loop

The main part of the program is a big loop that checks to see if: one second has passed and the real-time clock needs adjusting; the user has typed enough characters for one command; it is time to output a reading; or a character can be sent out to the terminal. The attempted outputting of characters every so often is known as "polling." It depends on the fact that the program is looping over and over again, and judiciously placed attempts at outputting data will allow the CPU to do something useful (i.e., output to the terminal) while waiting for user input or the completion of an A/D conversion. The entire software design is based on small functions that spend very little time working and put their results into an output buffer whose characters are periodically sent to the terminal.

As in all good programs, we

first initialize the hardware and storage areas. The PIA needs to be set up for various input and output lines. We then tell the ACIA how many data and stop bits to utilize, and set up flags and buffer pointers as appropriate. A sign-on message is loaded into the output buffer, part of it is output in a loop, then we enter the main program where the rest of the sign-on message is output as part of the regular polling operation. At this point the program loops, processing the real-time clock, accumulating user command input, taking readings and outputting them (if any have been scheduled), and attempting to output characters to the terminal if any remain in the output buffer. While this is happening, the CPU is being interrupted at a 601-Hz rate (the real-time clock) and the user may be typing characters on a keyboard that cause interrupts so the characters can be stored and interpreted as a command.

The flowchart in Figure 1 shows the various operations that are handled in the main routine. Most of the time, the program is looking for something to do. When you complete entry of a command, the program performs the appropriate action according to the command letter and values associated with it. Each command routine uses very little processor time, and feeds all program-generated output into one periodically polled output buffer. In this way, time-critical events can still occur with a high probability of being detected and handled. For example, the "R" command reads the specified channel by: selecting channel 0 (to get the temperature sensor reference voltage), reading the voltage, selecting the specified channel, reading its voltage, then selecting the original channel again (because the scheduler expects the correct channel to be ready). Now that a valid reading has been obtained, the channel's formatting information

sets up temperature conversion, decimal point location, **time-stamping parameters**, and places the value in the output buffer. This reading will be sent to the terminal along with any other buffered output when normal polling takes place.

The MAIN program also keeps track of the time spent between typed characters. When you are in the middle of entering a command, the program simply places the characters into an input buffer and doesn't act upon the line until it has been terminated with a Return. If you begin entering a command and delay more than 10 seconds between any two characters, the program cancels the line and reverts to its normal polled operation. This time-out feature deals with occasional line noise when using the A/D box with a modem.

The SCHED routine is com-

plex, dealing with the event tables that govern which channels will be read, how many seconds elapse between readings, upper and lower limits, and channel formatting data. The MAIN program has already stored this information in several tables that contain all of these parameters along with which channels will be processed. If a desired active channel is in one of these tables, then a fresh A/D sample is taken, the next channel is selected, and the value is formatted according to temperature conversion. The elapsed timer is **decremented** by 1 each time the scheduler is called, and if it gets to 0, the channel's value is output. If either an upper or lower limit is set, and the current value exceeds either limit, then the channel's value is output. By selecting the next channel immediately after reading the desired one, the A/D converter has

adequate time to sample its input before a new value is requested by the program, especially due to the extra reading that is necessary when the input voltage changes polarity. With 16 different inputs, the box has no idea of the voltage level it will have to convert as it scans from channel to channel.

The scheduler is governed by two very important data structures. One is an array of 16 channel parameters that contains lower and upper limits, time interval, time remaining, current value, and some individual bits that are set by the channel-specific commands in the MAIN program (listed in Table 2). All of the information that the scheduler requires to process a given channel is kept in this array. Upper and lower limits are checked only if they have been specified. The time remaining is counted down each second and reset to the

### Powerful, Low-Cost Data Acquisition and Control with Commodore C64 & C128



80-line Simplified Digital I/O Board with ROM cartridge socket  
Model SS100 Plus \$129. Additional \$119.



Original Ultimate Interface  
Universally applicable dual 6522 versatile interface adapter board.  
Model 64IF22 \$169. Additional \$149.

16-Channel, 8-bit analog-to-digital conversion module.  
Requires model 64IF22. Model 64IF/ADC0816 \$69.

Interface boards include extensive documentation and program disk. Manuals available separately for examination. Call or write for detailed brochure.

#### Resources for Serious Programmers.

- Symbol Master Multi-Pass Symbolic Disassembler. C64 & C128. \$49.95
- PTD6510 super-powerful Symbolic Debugger. C64. \$49.95
- MAE64 6502/65C02 Macro Editor/Assembler. \$29.95
- C64 Source Code Book. Kernal and Basic ROMs. \$29.95

### SCHNEDLER SYSTEMS

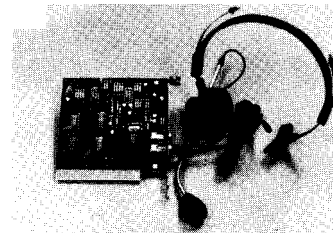
Dept. C, 25 Eastwood Road, P.O. Box 5964  
Asheville, North Carolina 28813 Telephone: (704) 274-4646

Circle No. 122 on Reader Service Card

## VOICE MASTER KEY<sup>®</sup> VOICE RECOGNITION SYSTEM

FOR IBM, PC, XT, AT AND COMPATIBLES

GIVES A NEW DIMENSION TO PERSONAL COMPUTING The amazing Voice Master Key System adds voice recognition to just about any program or application. You can voice command up to 256 keyboard macros. Requires under 64K. Instant response time and high recognition accuracy Works with CAD, desktop publishing, word processor, spread sheet, games, even other TSR programs! Voice Master Key can also be called from within a program for adding voice recognition to custom applications. A genuine productivity enhancer. Easy and fun to use-the manual has you up and running in under an hour! A price/performance breakthrough equal to other systems costing \$\$\$ more!



ALL HARDWARE INCLUDED  
Consists of a short plug in board that fits in any available slot. External ports include microphone and line level inputs. High gain flat response headset microphone included. High quality throughout.

ONLY \$129.95 COMPLETE

ORDER HOTLINE: (503) 342-1271 Monday-Friday, 8 AM to 5 PM Pacific Time

Add \$5 for shipping and handling on all orders. Add an additional \$3 for 2nd day delivery. All goods shipped UPS. Master Card and VISA, money order, cashiers check or personal checks accepted (allow a 3 week shipping delay when paying by personal check). Foreign inquiries contact Covox for C&F price quotes. Specify computer type when ordering. 30 DAY MONEY BACK GUARANTEE IF NOT COMPLETELY SATISFIED. ONE YEAR WARRANTY ON HARDWARE.

Call or write for FREE product catalog.



**COVOX INC.**

675-D Conger Street, Eugene, OR 97402  
Telex 706017 (AV ALARM UD)

TEL: 503-342-1271 FAX: 503-342-1283

Circle No. 109 on Reader Service Card

time interval when it gets to 0. Another array or list is generated by the INILIST subroutine every time a command is processed. This list contains the channel number of every channel that is currently active as defined by certain bits in the individual channel's parameter array. Once each second, the scheduler processes only those channels whose numbers are in this short list rather than all channels. This eliminates the delay waiting for the A/D conversion to finish on channels that have nothing connected to their inputs. Channel 0 is always entered in the list as the first and last channel for temperature conversion.

Another important routine, **GETAD**, actually reads and forms the value from the A/D chip. This routine is the only one that must actually spend time waiting for something. While it's waiting, however, it attempts to send characters out to the terminal. When the A/D chip has completed sampling the selected channel, its EOC (End Of Conversion) signal is detected and **GETAD** starts accepting digits. The first digit becomes the thousands digit. The hundreds digit is next, then the tens digit, and finally the units digit. Other status information accompanies the thousands digit and is dealt with after a value has been accumulated. If the sign bit is one, then the value is negated. If the value is zero and this is the first of two possible readings for this particular channel, then the process repeats to obtain a valid reading due to the change in polarity anomaly discussed earlier. If the overrange bit is one, a value of 9999 is returned. The digits are presented to the program at approximately 5000 per second, but even if the program misses one, they continuously repeat for the entire time between sampling. In actuality, the program operates fast enough to never miss a digit.

One of the first routines writ-

```

main()
{
    register struct CHAN * C;
    register int i;      /* general integer */
    int foo;             /* dummy variable */

    /* initialize hardware and software once only */
    inimem(1);          /* all variables */
    inilist();           /* channel list */
    inithw();           /* PIA and ACIA */
    foo = (ocount / 2) + 1;
    while (ocount >= foo)
        outc();          /* force half of string out */
    while (gchar() != -1)
        ;                /* empty input buffer */

loop:   /* main processing loop */

    /* handle real-time logging clock */
    if (seconds >= 60) {
        seconds -= 60; /* in case someone takes a long time */
        if (++minutes >= 60) {
            minutes = 0;
            if (++hours >= 24)
                hours = 0;
        }
    }

    /* output characters in buffer */
    outc();              /* output something if possible • /

    /* handle keyboard characters in buffer */
    if (icount)          /* characters in input buffer? */
        gline();         /* deal with them */

    /* handle timeout to resume scheduler */
    if (onsec > TIMEOUT) { /* waited long enough? */
        schar('^');      /* output ctrl-U */
        schar('U');      /* "^U" means line ignored */
        schar(CR);       /* output cr or cr/lf */
        lpos = lbuf;     /* reset pointer */
        onsec = inhibit = 0; /* resume */
    }

    /* handle scheduler */
    if (onsec AND !inhibit) /* clock requesting service? */
        sched();         /* do appropriate things */

    /* handle completed command line */
    if (eol) {           /* end of input line? */
        parse();         /* break it up */
        if (!(error)) { /* parsed successfully */
            c = (struct CHAN *) &ad[chnum];
            switch (cmdltr) { /* see what to do */
            case 'A': /* set clock at value */
                i = cmdval / 100;
                cmdval = cmdval - (i * 100);
                if (i >= 24 OR cmdval >= 60)
                    goto erred: /* invalid time value */
                else {
                    hours = i; /* set clock */
                    minutes = cmdval;
                    seconds = 0;
                }
                break:
            case 'C': /* crlf handling */
                crlf = cmdval;
                break;
            case 'D': /* decimal point */
                c->mode &= (-1-3);
                c->mode |= (cmdval & 3);
                break:
            }
        }
    }
}

```

Listing 1 -- (continued on page 27)



```

case 'E': /* echo handling */
    echo = cmdval;
    break;

case 'I': /* interval time */
    if (cmdval) /* value to use */
        c->mode |= IBIT;
    else /* no value, turn mode off */
        c->mode &= (-1-IBIT);
    c->tleft = c->intrvl = cmdval;
    break;

case 'L': /* low limit */
    c->mode |= LBIT;
    c->lolim = cmdval;
    break;

case 'O': /* once-only mode */
    c->mode &= (-1-OBITS);
    if (cmdval)
        c->mode |= OBIT;
    break;

case 'P': /* parity handling */
    parity = cmdval & 3;
    break;

case 'R': /* read a/d */
    i = select(0); /* select zero */
    getad(); /* bogus reading */
    getad(); /* channel zero value */
    ch0val = advalue;
    select(chnum); /* select new */
    getad(); /* bogus reading */
    getad(); /* true value */
    select(j); /* select previous */
    if (i = (c->mode & TBITS))
        dotemp(i); /* convert for temperature */
    log = (c->mode & WBIT);
    dot = (c->mode & DOTS);
    outad(chnum,advalue); /* show value */
    break;

case 'S': /* status */
    c = (struct CHAN *) &ad[0];
    sumdig = 1; /* flag for active channels */
    for (i = 0; i < NUMCHAN; ++i) {
        if (c->mode & SBITS) {
            dostat(c,i);
            sumdig = 0; /* got one */
        }
        ++c; /* next */
    }
    if (sumdig) { /* anything? */
        schar('N');
        schar('o');
        schar('n');
        schar('e');
        schar(CR);
    }
    break;

case 'T': /* temp mode */
    c->mode &= (-1-TBITS);
    cmdval &= 3;
    if (cmdval & 1) /* centigrade */
        c->mode |= CBIT;
    if (cmdval & 2) /* fahrenheit */
        c->mode |= FBIT;
    break; /* both on means kelvin */

case 'U': /* upper limit */
    c->mode |= UBIT;
    c->hilim = cmdval;
    break;

```

Listing 1 -- (continued on page 28)

## THE INTERCHANGE™

Bi-directional Data Migration Facility  
for IBM PS/2, AT, PC, PORTABLE  
and Compatibles

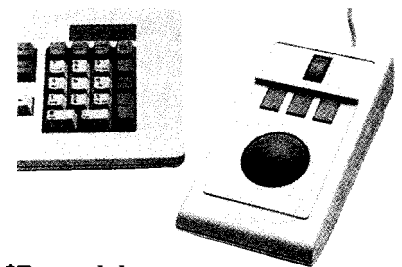


### Features:

- \*Parallel port to parallel port.
  - \*Economical method of file transfer.
  - \*Bi-Directional file transfer easily achieved.
  - \*Supports all PS/2 systems (Models SO, 50, 60, and 80).
  - \*Supports IBM PC, XT, AT, Portable and 100% compatibles.
  - \*Supports 3 1/2 inch and 5 1/4 inch disk transfers.
  - \*Supports hard disk transfers.
  - \*Supports RAMdisk file transfers.
  - \*The SMT 3 Year Warranty.
- ONLY \$39.95**

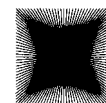
## FastTrap™

The pointing device of the future is here!



- \*Two and three axis pointing capability.
- \*High resolution trackball for X and Y axis input.
- \*High resolution fingerwheel for Z axis input.
- \*Use with IBM® PC's, XT's, AT's and compatibles.
- \*Three input buttons.
- \*Full hardware emulation of Microsoft® Mouse.
- \*Standard RS-232 serial interface.
- \*Includes graphics drivers and menu generator.
- \*Easy installation.
- \*1 year warranty.
- \*Made in U.S.A.

**ONLY \$149.00**



**LTS/C Corp.**  
167 North Limestone Street  
Lexington, Kentucky 40507  
Tel: (606) 233-4166

Orders (800) 872-7279  
Data (606) 252-8968 [3/12/2400 8-N-1]  
VISA, Mastercard, Discover Card,  
TeleCheck

Circle No. 117 on Reader Service Card

ten was PARSE, the user command parser. This routine separates the typed input line into its various parts and validates each component. PARSE deals directly with the user's input and is very stringent and demanding on exactly what it will let squeeze through. The rest of the software expects good data, so this routine has to ensure that valid commands pass and invalid commands don't.

A command is formed from numbers and letters. You enter a string of optional digits specifying the channel number from 0 to 15, followed by a letter, which can be upper or lower case specifying the actual command to be performed, then another sequence of optional digits which may be preceded by a hyphen to indicate a negative value, and terminated with the Return key. For example, 3 15, 10 L 1500, 10 U -1674, 8 D 3. Blanks may be inserted for readability but they will be ignored. Since 16-bit integers are used for variables within the program, the values must be between 0 and 65535. In the computer's numbering system, 65535 is equal to -1, 65534 equals -2, and so on. To make the box more human compatible, a negative number is accepted and converted to its equivalent 16-bit value. This value is used by some commands where a value is significant; it is ignored by other commands. Any characters other than numbers, letters, and hyphens are eventually stripped from the input data. When you terminate the line by pressing Return, the command is broken down into its three components. If any of the numeric parts are missing, they will default to a value of zero. Any input line that does not meet the above format criteria is ignored and an error indication consisting of a zero value for channel 99 is output to the terminal as 99 0000. If the value presented exceeds the usable range of the particular function, then undeter-

```

case 'W': /* whether to log or not */
    c->mode &= (-1-WBIT);
    if (cmdval)
        c->mode |= WBIT;
    break;

case 'X': /* miscellaneous functions */
    if (cmdval == -1)
        inimem(0); /* reset all channels */
    else if (cmdval == -2) {
        c = (struct CHAN *) &ad[0];
        for (i = 0; i < NUMCHAN; ++i) {
            c->tleft = 1; /* synchronize */
            ++c; /* next */
        }
    }
    break;

case 'Z': /* zero mode, time, limits */
    c->tleft = c->intrvl = c->mode = 0;
    c->lolim = c->hilim = c->curval = 0;
    break;
) /* end switch */
inilist(); /* update channel list */
} else {
erred:
    log = dot = 0; /* output as 0000 */
    outad(99,0); /* channel 99 is error */
    inhibit = eol = 0; /* accept another line */
}
got0 loop: /* loop until power turned off */
} /* end main() */

```

Listing 1 -- (continued from page 27)

```

getad() /* get a/d reading */
{
    register char * p; /* -> port */
    register int bcd; /* BCD digit */
    int status: /* negative, overrange */
    int zflag: /* zero reading flag */
    int toc; /* timeout counter */

    p = (char *) PIA; /* setup pointer */
    zflag = 1;

again:
    toc = -1; /* set timeout delay */
    bcd = p[2]; /* dummy read clears flag */
    while (!(p[3] & 0x80)) { /* MSB = end of conversion */
        outc(); /* output something while waiting */
        if (--toc == 0) {
            advalue = 8888; /* timed out waiting for ready */
            return; /* that's all, folks */
        }
    }

    while (!(status = p[2]) & 0x80)
        ; /* wait for thousands digit */

    advalue = (status & 0x08) ? 0 : 10; /* (0 or 1) * 10 */

    while (!(bcd = p[2]) & 0x40)
        ; /* wait for hundreds digit */
    advalue += (bcd & 0xF); /* merge it in */
    advalue *= 10;

    while (!(bcd = p[2]) & 0x20)
        ; /* wait for tens digit */
    advalue += (bcd & 0xF); /* merge it in */
    advalue *= 10;
}

```

Listing 2 - (continued on page 29)

```

while (!(bcd = p[2]) & 0x10)
; /* wait for ones digit */
advalue += (bcd & 0xF); /* merge it in */

if ((status & 0x09) == 1) /* overrange? ● /
advalue = 9999; /* set error value */
if (!advalue AND zflag) { /* if first reading zero */
zflag = 0; /* set for final reading */
got0 again; /* and do it again */
}

if (!(status & 0x04)) /* negative polarity? */
advalue = -advalue; /* invert value */
} /* end getad() */

```

Listing 2 -- (continued from page 28)

```

parse() /* parse command line */
{
register char * p: /* -> line buffer */
register char * q: /* -> line buffer */
int sign; /* value sign */

p = q = lbuf; /* prepare to verify */
while (*p) {
if (*p == '-' OR (*p >= '0' AND *p <= '9') OR
(*p >= 'A' AND *p <= 'Z'))
*q++ = *p; /* only allow these characters */
++p;
}
*q = '\0'; /* null-terminate */

p = lbuf; /* setup pointer */
cmdval = chnum = error = 0; /* init variables */

while (*p >= '0' AND *p <= '9') /* channel number is first */
chnum = chnum * 10 + (*p++ - '0');

if (chnum >= NUMCHAN) /* invalid channel number? */
error = 1; /* seems to be */

if (*p) /* then command letter */
cmdltr = *p++; /* was there a command? */
else
error = 1; /* apparently not */

'sign = 0; /* presume positive value */
if (*p == '-') /* minus sign? */
sign = *p++; /* remember it */

while (*p >= '0' AND *p <= '9') /* then command value */
cmdval = cmdval * 10 + (*p++ - '0');

if (sign) /* negative number? */
cmdval = -cmdval; /* negate result */
} /* end parse() */

```

Listing 3

mined results may occur.

A typical command would be 3120 which says to set channel 3 to Interval mode and take a reading every 20 seconds. If the command was entered as 130 then channel 0 would be set to Interval mode and

a reading would be taken every 30 seconds. To stop the Interval mode operation, a value of 0 is given, as in 310 or 31 since a missing number defaults to zero.

While you are entering a command, the characters are placed

into an intermediate buffer where **rubout** or backspace character editing can be applied. When Return is detected, the intermediate buffer is cleaned up and passed to the command parser which separates the components. Lower-case characters are converted to upper case and nonprintable characters are eliminated. The contents of this buffer are further cleaned up by the command parser when a complete line has been entered.

### A Subroutine for Every Job

The subroutines can be divided into classifications such as once-only, input or output formatting, interrupt, and general. Descriptions of each subroutine may be found in Table 1. A few of the key routines are included in this article. If you have downloaded the entire program, you can follow all of the descriptions in detail.

The rest of the source file contains necessary equates and addresses required for a ROM program to interface to the M6809 architecture. These values are stored at hex addresses FFF6 through FFFF as vectors.

### A Comprehensive Command Set

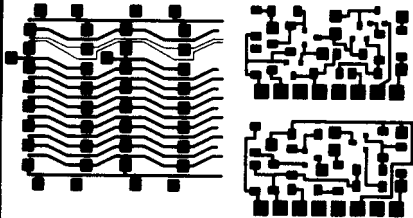
The A/D box has a simple, concise command entry format as described above. Since there are only 26 letters available for commands, I attempted to make as many as possible memorable by using the first letter of the command word as the actual command. The commands can be split into three groups: communication parameters, initial setup, and channel-specific commands. The channel-specific commands manage data in the channel parameter array which eventually governs the scheduler's operation. The complete list of commands is shown in Table 2.

If you build your own data logger from the information in this

# DROE GE

DESIGN ROBOT FOR  
THE ORIGINATOR OF  
EXACTING GRAPHIC  
ENGINEERING

A MANUAL PRINTED  
CIRCUIT CAD/CAM  
SYSTEM



MULTI-LEVEL SYMBOL CONCEPT. MEMORY LAYOUT ABOVE IS A SYMBOL MADE FROM TWO CHIP SYMBOLS WITH ADDED BUS WIRE! CHIP SYMBOLS ARE MADE FROM MULTIPLE PAD SYMBOLS.

- BASIC \$10.00 POSTPAID:**
- CGA 3 COLORS 12 LAYERS
  - 64 BY 64 INCH WORKSPACE
  - ANY GRID TO 0.001 INCH
  - RUNS ON ANY PC COMPATIBLE
  - 15 LINE WIDTHS
  - DOT MATRIX OUTPUT
  - 200KB DOCUMENTATION ON TWO DISKS
  - 16 WORK AREAS SAVEABLE
  - STITCH BETWEEN LAYERS
  - ZOOM AND PAN TO ANY SCALE
- ADVANCED \$6100.00 POSTPAID:**
- ABOVE FEATURES
  - EGA RESOLUTION 15 COLORS
  - LARGER JOBS
  - MOUSE
  - ADVANCED EDITING
  - PRINTED MANUAL

ENVIRONMENTAL OPTICS CORP.  
P.O. BOX 296 BATAVIA, IL 60510

SEND BASIC  ADVANCED  INFORMATION   
 CHARGE VISA  MC  PAYMENT ENCLOSED   
 CHARGE NUMBER \_\_\_\_\_ EXP \_\_\_\_\_

NAME \_\_\_\_\_  
 ADDRESS \_\_\_\_\_  
 CITY \_\_\_\_\_  
 STATE \_\_\_\_\_ ZIP \_\_\_\_\_

TELEPHONE ORDERS EVENINGS (312) 879-2949  
 THIS AD WAS COMPOSED WITH DROE GE AND  
 PLOTTED ON OUR PHOTO PLOTTER. WE CAN MAKE  
 PHOTO PLOTS FOR YOU FROM PROGRAM OUTPUT.

```

sched()          /* one-second reading processing */

register struct CHAN * c; /* -> a/d parameters */
register char * chptr; /* channel pointer */
int chanel; /* current channel */
int tb; /* temp bits */
char * debug; /* DEBUGGER */

debug = (char *) PIA;
debug[1] = 0x3C; /* set DEBUGGER */

chptr = chlist; /* -> channel list */

while (chptr[1] >= 0) { /* stop when hit -1 terminator */
  if (ocount >= FULLBUFF) /* room in buffer? */
    break; /* can't do anything else if not */

  chanel = *chptr; /* current channel */
  c = (struct CHAN *) &ad[chanel];
  getad(); /* take a reading */
  select(chptr[1]); /* select next channel */

  if (chptr == chlist) { /* temp cal? */
    ch0val = advalue; /* save it */
    goto skipit; /* done with it */
  }

  if (tb = (c->mode & TBITS)) /* temp? */
    dotemp(tb); /* adjust temp */

  c->curval = advalue; /* remember new value */

  dot = (c->mode & DOTS); /* set dot position */
  log = (c->mode & WBIT); /* set logging option */

  if (c->mode & IBIT) { /* interval mode on? */
    if (--c->tleft == 0) /* time to take reading? */
      c->tleft = c->intrvl; /* reset time left */
    outad(chanel, advalue); /* output value */
    goto skipit;
  }

  if (c->mode & ULBITS) { /* limit check? */
    if (((c->mode & UBIT) AND advalue > c->hilim) OR
        ((c->mode & LBIT) AND advalue < c->lolim)) {
      if (((c->mode & OBIT) == 0) OR
          ((c->mode & OBITS) == OBIT))
        outad(chanel, advalue);
      c->mode |= OLBIT; /* out of limits now */
    }
    else {
      c->mode &= (-1-OLBIT); /* not out of limits */
    }
  }

  skipit:
  ++chptr; /* on to next channel in list */
}
onsec = 0; /* we did one pass */
select(0); /* reselect first channel */

debug[1] = 0x34; /* clear DEBUGGER */
} /* end sched() */


```

Listing 4

article, I would expect the parts to cost between \$50 and \$100.

### Acknowledgements

I wish to thank and acknowledge the assistance of Leo Taylor. He did all of the hardware design

and is the current user and owner of the prototype A/D box at his home. I designed and wrote most of the software, with much help from Leo regarding communication with the ACIA and PIA. 

### IRS

- 207 Very Useful
- 208 Moderately Useful
- 209 Not Useful

MAIN, shown in Listing 1, is one big loop that processes the user's requests. All routines except interrupt service routines are called from MAIN or from routines that were called by MAIN.

ASCDIG performs some of the operations for converting a binary value into printable ASCII digits. It uses repetitive subtraction rather than division. This is one of the output formatting functions.

DOCHAN outputs the channel number by calling ASCDIG. As with all output routines, the information is stored in the output buffer.

DOSTAT is called by the status command for each channel to be displayed. It checks the various bits and values saved in the channel parameter area and outputs a translation of their meaning.

DOTEMP handles the conversion of a channel's value according to any temperature format desired. The channel 0 reading (approximately -1.8 volts) is subtracted from the current channel's value in this routine which results in a Kelvin temperature in tenths of a Kelvin. Appropriate formulas convert this to a Celsius or Fahrenheit temperature. This new value is then stored as the current channel's value so limits can be compared in the scheduler.

DOVAL outputs the channel's value by calling ASCDIG with various powers of 10. It also places the decimal point where appropriate. As with DOCHAN, the results are put in the object buffer.

GCHAR removes any characters from the input buffer. The input buffer is filled when the user enters characters which cause interrupts from the ACIA and subsequent processing.

GETAD, shown in Listing 2, reads the A/D converter and formats the raw data into an integer value. The routine takes two readings if the first is zero. It also checks for overrange and polarity.

GLINE builds a line of input from characters typed by the user. It detects end-of-line characters (CR and LF), handles rubout and backspace, and echoes printable characters.

INILIST makes a list of requested A/D channels according to various bits set by the user's commands. This list tells the scheduler which channels to process, and is updated after each command processed in MAIN.

INIMEM initializes all memory locations used by the program. It also sets up the copyright notice that is seen when the A/D box is first turned on. This is a once-only routine.

INITHW initializes the PIA and ACIA when the A/D box is turned on. This is also a once-only routine.

OUTAD calls DOCHAN and DOVAL to output a reading from the A/D box. It also appends the real-time clock's time if desired.

OUTC outputs characters if the ACIA is ready to accept another character and the user hasn't typed Ctrl-S to suspend output.

OUTLINE stores a null-terminated line of information in the output buffer. If there is insufficient room in the buffer, it loops while attempting to send characters to the ACIA. This is one place where readings can be dropped while waiting for room in the output buffer.

PARSE, shown in Listing 3, copies characters from the input buffer to the intermediate buffer, passing only numbers, letters, and hyphens. It then separates this line of input into its various components and verifies the channel number.

PBIT adds a parity bit to an output character according to your specifications. It actually counts the bits that are on in the character. While a table lookup might have been faster, this was more fun to write.

SCHAR stores characters (with parity added) into the output buffer and terminates each line with CR and an optional LF.

SCHED, shown in Listing 4, handles the one-second reading of all specified channels. It reads the channel and adjusts the result according to the channel 0 value and temperature conversion specified, checks to see if the channel should be output at this time interval, checks to see if it is above or below either of the two limits, and sets up other parameters as required for formatting the value. The main controlling logic for automatic operation is in this routine.

SELECT converts the channel number to the appropriate bit configuration necessary to select one of the two data multiplexers as well as the channel on that multiplexer. It also saves the original channel number for times when it must be restored.

CINTR is an interrupt service routine that counts clock pulses. When 601 have occurred, it resets the pulse counter and increments a one-second counter that gets checked in the MAIN routine.

IINTR is an interrupt service routine that gathers characters from the ACIA. It strips the parity bit, deals with Ctrl-S by setting a flag that will prevent further characters from being output, and stores all other characters in the input buffer. These characters are processed by GCHAR.

RESTART is the assembly language routine which initializes the CPU and then calls MAIN, which never returns.

CCMULT and CCDIV are assembly language routines that respectively multiply and divide two 16-bit integer values using shift and rotate instructions. This method ensures completion in a finite time regardless of the magnitude of the values being calculated.

IRQ is the actual code that the CPU executes when it processes a maskable interrupt from the ACIA. This assembly language routine only calls the IINTR routine.

NMI is the actual code that the CPU executes when it processes a nonmaskable interrupt from the 601-Hz clock. This assembly language routine only calls CINTR.

**able 1 -- The data logging software is very modular, with each subroutine performing a specific function.**

# INTROL CROSS DEVELOPMENT SYSTEMS

- INTROL-C Cross-Compilers
  - INTROL-Modula-2 Cross-Compilers
  - INTROL-Macro Cross-Assemblers
- Provide cost and time efficiency in development and debugging of embedded microprocessor systems

All compiler systems include:  
Compiler • Cross-assembler • Support utilities • Runtime library, including multi-tasking executive • Linker • One year maintenance • User's manual, etc.

TARGETS SUPPORTED:  
6301/03 • 6801/03 • 6804 • 6805 • 6309  
• 68HC11 • 68000/08/10/12 • 32000/  
32/81/82 • 68020/030/881/851

AVAILABLE FOR FOLLOWING HOSTS  
VAX & MicroVAX; Apollo; SUN; Hewlett-Packard; Gould PowerNode; Macintosh; IBM-PC, XT, AT and compatibles

INTROL CROSS-DEVELOPMENT SYSTEMS are proven, accepted, and will save you time, money, effort with your development. All INTROL products are backed by full technical support. CALL or WRE for facts NOW.



**I N T R O L  
C O R P O R A T I O N**

647 W. Virginia St., Milwaukee, WI 53204  
414/276-2937 FAX: 414/276-7026  
Quality Software Since 1979

Circle No. 114 on Reader Service Card

## Communication Parameters Commands

**C** - (Default) output a **linefeed** as well as a Carriage return at the end of each line. **C** is enabled by any **nonzero** value and disabled by a zero. The channel number, if given, is ignored.

**E** - (Default) Echo the input characters back to the terminal. Enabled by any **nonzero** value, disabled by a **zero** value. The channel number, if given, is ignored.

**P** - set Parity according to the value given. Default is **zero** parity (i.e., the parity bit is always set to 0). A value of 1 sets the parity bit to **1**. A value of 2 sets the parity to even. A value of 3 sets the parity to odd parity. The channel number, if given, is ignored. The parity of all input characters is ignored by the A/D box.

## Initial Setup Commands

**A** - set internal time-stamping clock time to the value given as hours and minutes. Time is entered in 24-hour format with 1 or 2 digits for hours, and 2 digits for minutes. The channel number, if given, is ignored. The time clock defaults to 0 hours and 0 minutes when the box is turned on.

**S** - show Status of all channels. Both the channel number and value, if given, are ignored. All active channels' parameters are displayed as:

```
cc I = iiii L = 1111 U = uuuu t 0 W
```

where "cc" is the channel number from 00 to 15; "I = iiii" indicates the interval time in seconds; "L = llll" indicates the low limit; "U = uuuu" indicates the high limit, both values displayed according to the decimal point selection; "t" will be either C, F, or K depending on the temperature conversion in effect; "O" indicates that once-only mode is specified; and "W" means that time stamping is to be displayed. The "O" can come out as "OO" if the once-only mode is in effect and the reading is currently out of range and has been displayed its one time. Only the currently active parameters will appear on the status line.

**X** - special functions. The channel number, if given, is ignored. If the value is 65535 or -1, then ALL channels' parameters are reset to their power-up conditions. If the value is 65534 or -2, then ALL channels' interval timers are **synchronized** so they will all occur at the same time if they all have the same interval specified.

## Channel-Specific Commands

**D** - set Decimal point after 1st, 2nd, or 3rd digit according to the value given. Normally, readings are output as a number of millivolts, but you can display it as volts by setting **D1** on the particular channel. If you have installed a 10:1 divider network on any channel, setting **D2** will display up to  $\pm 19.99$  volts. Setting **D0** will remove the decimal point entirely.

**I** - set Interval time between readings in seconds. This will automatically set up the specified channel for periodic readings by the scheduler. If the value given is 0, then interval mode is turned off. For example, entering 3110 schedules channel 3 to take a reading every 10 seconds. The interval mode, if enabled, will produce output regardless of any upper or lower limits.

**L** - set the Lower limit to the specified value. A reading that is lower than this will be displayed, one that is not will not be displayed. The "O" command can alter this behavior. The value may be entered with a decimal point (which will be ignored) if desired.

**0** - display reading Once-only when it exceeds either the lower or upper limit. This avoids the voluminous amount of output that would be generated when a channel's reading is out of limit. When the sampled voltage goes back within the limit range, the channel is reset such that it will be displayed when it again exceeds either limit. Any value, if given, is ignored.

**R** - Read the specified channel once immediately. Any value given is ignored. The output is formatted according to the "D", "T", and "W" commands in effect at the time for that channel. The channel parameters are NOT modified by this command. The particular channel may be read several times by the program before a valid reading is obtained.

**T** - set Temperature conversion mode. A value of 0 specifies normal voltage readings. A value of 1 converts it to temperature in degrees Celsius. A value of 2 converts it to temperature in degrees Fahrenheit. A value of 3 converts it to temperature in Kelvin. This command only performs the appropriate conversion math for the output value. Setting the decimal point after the 3rd digit (with **D3**) is appropriate when displaying temperatures.

**U** - set the Upper limit to the specified value. A reading that is higher than this will be displayed, one that is not will not be displayed. The "O" command can alter this behavior. The value may be entered with a decimal point (which will be ignored) if desired.

**W** - enable time stamping or logging. The current time as set with the "A" command is appended to the reading displayed. If the time has never been set, then the current time is the number of hours and minutes since the box was turned on or last reset. Any value, if given, is ignored.

**Z** - reset all parameters for the specified channel. This clears any interval times, lower and upper limits, once-only mode, decimal point, and temperature conversion to their power-up values. Any value, if given, is ignored.

**Table 2 -- The complete command set for the data logger gives you control over most of the important I/O, processing, and conversion features of the system.**

# A Call for Dedication

by Ezra Shapiro

# INK SPOT

---

**M**any years ago, I worked for the Forest Service in Oregon. It was a physical, dirty job, slogging overland into the wilderness to survey for logging roads. Our clothing reflected the realities of this life; we wore work boots, blue jeans, wool shirts, and hard hats. We chuckled at the weekend hikers who wandered up and down the trails clad in hundreds of dollars' worth of bright rip-stop clothing. We knew that branches, thorns, and rocks could easily shred their flimsy garments; you wouldn't catch us spending our money on clothing we knew would be destroyed.

We called these people "equipment kings," and I've noticed the phenomenon repeated in dozens of settings. Nowhere is it more common than in the computer business. We all know pencil-pushers who purchase killer '386 systems to write simple memos, the latest version of dBASE to organize Rolodex data, and humongous hard disks to schedule appointments with SideKick. Pretty funny, huh? Not really.

A growing percentage of computer users have reached the point where they know their needs and prefer efficient computer solutions to overkill. They're frustrated with an industry that demands conspicuous consumption. Ever watch the face of a user who's just been told what switching to OS/2 will cost? It's not a pretty sight.

I believe these people represent a small but expanding market for new approaches to computing. I receive amazed stares when I explain that although I use a state-of-the-art Macintosh system for desktop publishing, the rest of my work is done on a collection of oddball machines: two laptops and a Canon Cat, none of which runs MS-DOS or UNIX or OS/2. After a couple of dumbfounded questions, my listeners realize that I'm not crazy, and more than a few of them get very excited as they comprehend that they need not be trapped into what their dealers are pushing.

All three of my systems are text-processing environments with telecommunications capability, calculating or spreadsheet functionality, and some form of card-filing or database management. The CPUs are different, but all share a common trait: software in ROM. The Canon Cat boasts a flexible word processor created entirely in Forth. The NEC PC-8500 is a laptop with CP/M and WordStar in ROM. And my Cambridge Z88 has a Z80-based operating system that offers "lazy concurrency," an implementation of context-switching that allows me to open multiple programs by using RAMdisk to simulate RAM. With a couple of cables, a null modem, and a gender changer or two, all the machines talk to each other quite nicely.

Any of these is quite satisfactory for my work as a freelance writer. They are dedicated machines, though they bear little resemblance to the Wang and Lanier word processors of yesteryear. However, like their cumbersome predecessors, my dedicated machines will never be obsolete, because they're good at what they do. They aren't "development platforms," they're workhorses. Many consumers are moving down this path as well, though they're largely unaware of what they're doing. The sales rep who equips a laptop with Word Perfect Executive and nothing else, the internal publishing department that buys a Macintosh for PageMaker, the programmer who chooses a language package because of the integrated editor and debugger--all of them are building self-contained environments.

With today's PROM and PAL technology, these environments could easily be encapsulated into dedicated computers optimized for specific functions. It's a short step to dedicated database engines, portable tools for business executives, powerful analytical devices built around the spreadsheet paradigm, and more. While I don't see dedicated machines taking the market away from general-purpose computers, there are obvious niches begging to be filled.

So, you wanna make some money? All it takes is a good concept and a little dedication.

***Ezra Shapiro is a free-lance writer and publications designer based in southern California.***

# The Home Satellite Weather Center

## Part 7

by Mark Voorhees

### *Finishing the Firmware for the 68000 Peripheral Processor*

In the last few months, I've covered a lot of ground, and we're still a ways from the completion of the Home Weather Center. This time I'll cover the remainder of the 68000 Peripheral Processor firmware and provide an overview of an interface for the Heath ID-4001 Weather Computer (for those planning to use that unit as their instrument package).

In the last issue, I discussed concepts for most of the operating firmware for our Peripheral Processor (PP). I'll continue the discussion now with the routines involved in WEFAX processing. Some of this may not be clear now since I haven't presented the WEFAX hardware yet. I'll refer back to this discussion and review the highlights when we build that part of the project.

#### WEFAX Routines

As explained last time, WEFAX signals are given highest priority because we can't control their transmission. Several factors come into consideration when planning for the processing of WEFAX:

-The original picture consists of 800 pixels (picture elements) horizontally and 800 lines vertically.

-The original picture is essentially monochrome analog video at 4 Hz.

-Special signals are sent to indicate start-of-frame and end-of-frame; a 1-Hz "Sync Pulse" can also be decoded from the signal.

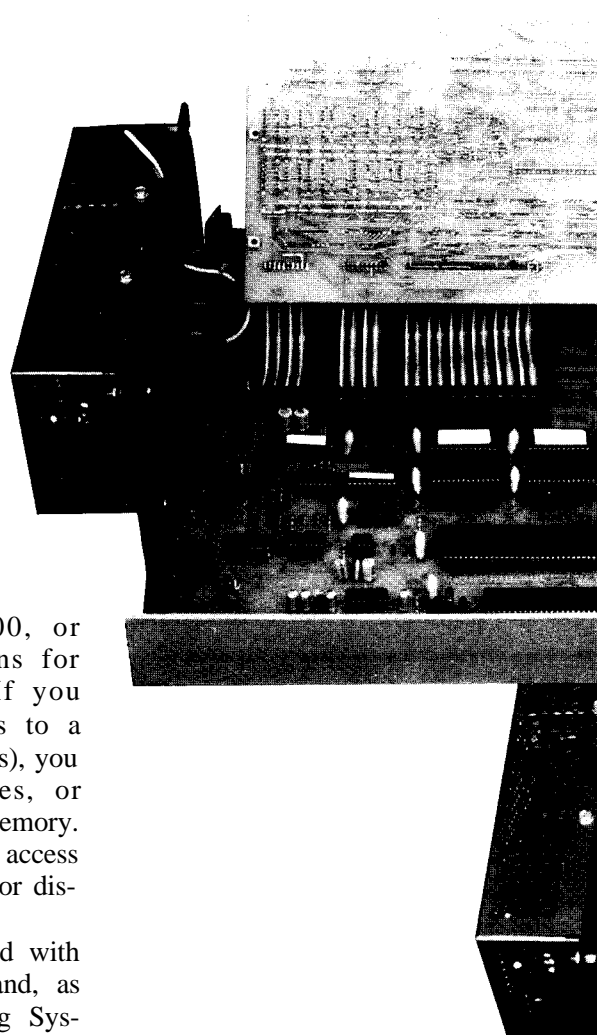
You need to do some advance work to handle memory assignments for the WEFAX data. The size specifications of WEFAX pictures determine our processing approach.

If you sample the signal based on the actual specification, you need approximately 800 x 800, or 640,000, memory locations for storage of the image. If you sampled all 640,000 points to a resolution of 8 bits (256 levels), you would need 640,000 bytes, or 320,000 words, of DRAM memory. At that rate you could only access the two most recent images for display.

Purists would be satisfied with this level of performance, and, as the Weather Data Processing System software expands, it may make use of this configuration. For now, however, it is a bit beyond the software's ability to display on a PC's graphics card. Let's look to save some space while maximizing display quality.

You can really only display a 640- x 200-line, 16-color image on a graphics card at this point, so

make 640 samples/line the sample rate (you'll see that the hardware will appreciate this as well). Also, only four bits are needed to provide 16 levels of gray. Memory usage has been dramatically reduced since 640 samples times 800 lines equals 512,000 memory locations, and with each holding four bits, we





need 256,000 bytes or 128,000 words of memory.

Allowing for memory used for instrument data, one megabyte of DRAM can hold six WEFAX images, with eight more in the second (optional) megabyte.

I'll use this configuration as the basic example for the firmware, although I'll allow for different configurations in the Weather Data Processing System software, with the sampling information made part of the configuration block.

You need to sense the frame start information, the "line sync" signal, the timing of the samples, and the termination, or ending, signal.

The hardware will be providing a status byte with the start, sync, and end signal sensing information developed in hardware, so that part will be relatively easy. The start signal will begin the sampling process, starting the memory save at a defined division; the sync

signal will define a line start address; and the end signal will end the process.

I'll use Timer A of the 6890 I on the CPU card to generate interrupts for sample starts. This is a crystal-clocked timer set for the sample rate at configuration time. The interrupt is enabled only after a frame start signal is received, and the timer is reset at each sync pulse. Thus, our samples should be as uniform as possible (some signal tracking is taking place in the hardware to help with stability of frequency).

There are two steps to programming the timer for the proper sample rate: the selection of a prescaler, and the count data.

The sample time will be defined as line rate divided by the number of samples per line. Inserting numbers, we have 250 ms per line divided by 640 samples per line, giving one sample every 390.625 microseconds.

Now, since the clock frequency is 2.4576 MHz (a period of 407 ns), the timer is set to interrupt every 960 clock pulses (nominal).

(For the "math wizards" among us: working backwards from our "convenient figure," you'll notice that the actual sample count ends up to be 639+ samples per line. Since the line length, of necessity, includes the sync pulse, which is unusable video, we'll accept this condition. In reality, the missing sample won't even be noticed.)

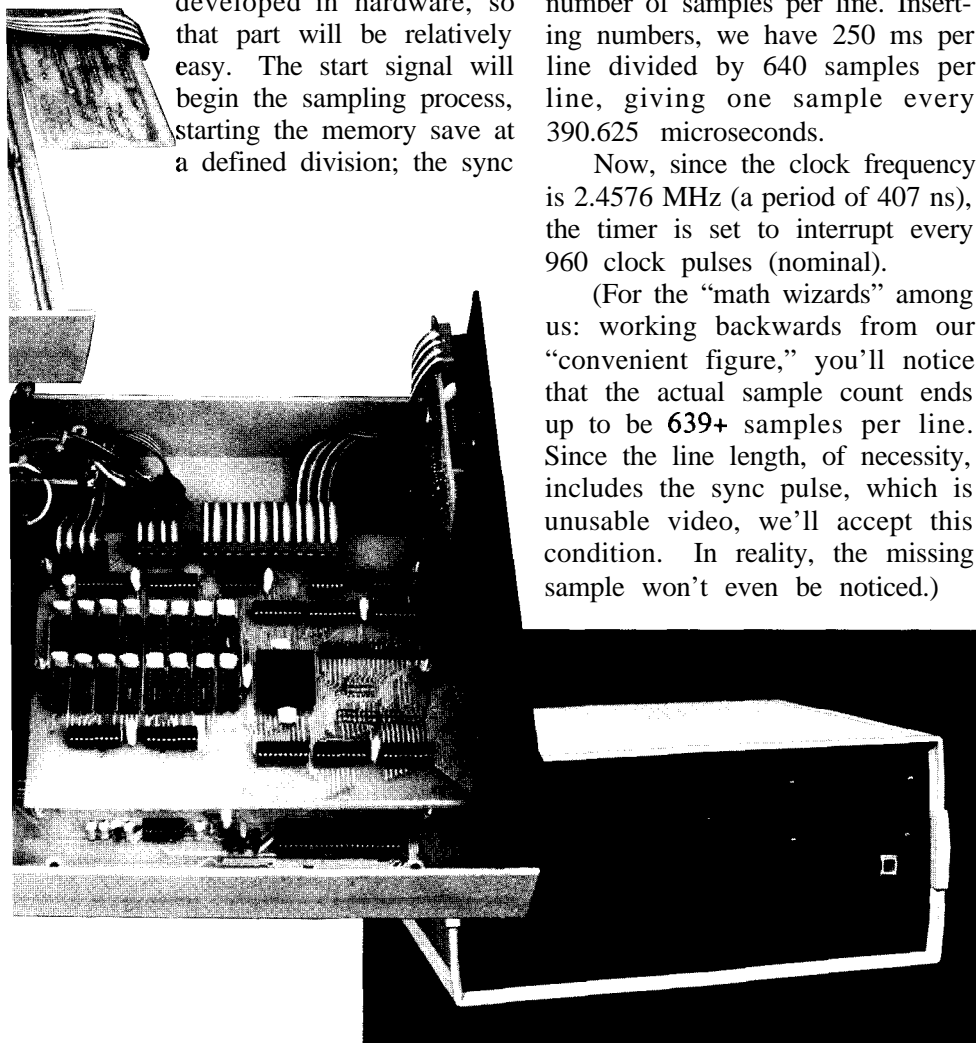
The timer provides for prescaling values of (divide by) 4, 10, 16, 50, 64, 100, and 200, as well as an S-bit count data byte, so it will easily meet our needs.

A count of 960 is convenient in another way. Since it is an even multiple of eight, several different prescaler and data byte combinations can toggle the interrupt. Looking ahead to future needs, when you might sample either 320 or 1280 times per line, you will have the most control by using the divide-by-16 prescaler. The data word then becomes 3C hex for current needs, 1E hex for the 320 samples per line possibility, and 78 hex for the 1280 samples per line possibility; all are within the 8-bit data byte limitation.

Once the PP receives the interrupt, the service routine will have to perform only a few tasks. First, immediately sample the WEFAX signal (to maintain the most even sample rate, this must be done first whether we need the sample or not). Next, check to see if the sync-pulse or end-of-frame flags have been set by hardware. If so, save the most recently processed sample and perform routines to advance to the next line (or reset for the next frame) before returning to the interrupted routine. If the flags are not set, save that recent sample we mentioned, prepare addressing for the next memory location, and return from the interrupt.

You'll see in a future installment that the processing hardware provides 8-bit digitized samples. Once again, this gives latitude for future enhancements. For now, I'll use the four most-significant bits for the 4-bit (16-level) sample.

The WEFAX service routine builds the 16-bit-wide memory word in "scratchpad" static RAM using bit rotations to place the "nibble" in its proper position. (Note that the PC's processing program will take this data after it's downloaded and disassemble it for



use in the graphics card).

There is one other "housekeeping" byte to manipulate during setup of the system. The WEFAX processor can get its demodulated audio from one of two sources: the integral receiver, which I'll present in a future article, or an external source. The software switches from one unit to the other by writing the "housekeeping" byte, and monitors for the presence of an audio carrier by reading the byte.

The only other area to be concerned with is the WEFAX receiver control, and this will be essentially a set-and-forget situation. The integral receiver receives on 137.5 MHz and 137.62 MHz, switched by the control byte. An AGC-voltage-monitoring circuit will signal the system via a status byte and interrupt if RF signal is lost. The RF loss will cause any WEFAX picture in progress to be discarded (it

wouldn't be complete), and inhibit any further processing until signal is restored.

That pretty much covers the principles of the firmware package. From time to time I may find it necessary to expand on the discussion in certain areas, and I'll also provide you with enhancement information as it becomes available.

[Editor's Note: Software and EPROM listings for this article are available from the Circuit Cellar BBS or on Software on Disk #7. For downloading and ordering information, see page 62. In addition, assembly listings for this article are available from the author for \$6.00. See the sidebar accompanying this article for more information.]

### Peripheral Processor Startup and Initial Testing

Rather than require a compli-

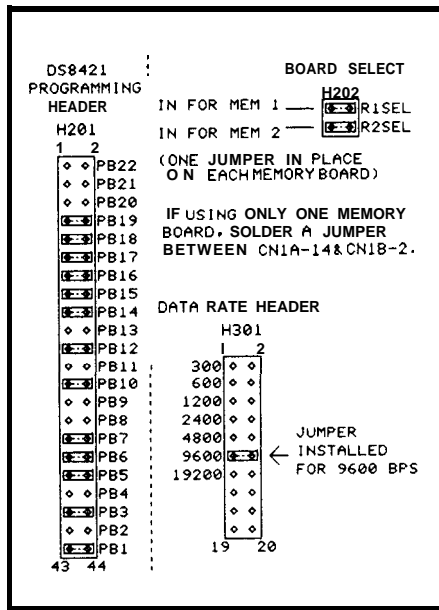


Figure 1 -- Jumpers for DRAM Memory Board, and Port Card jumper for default ID-5001 data rate.

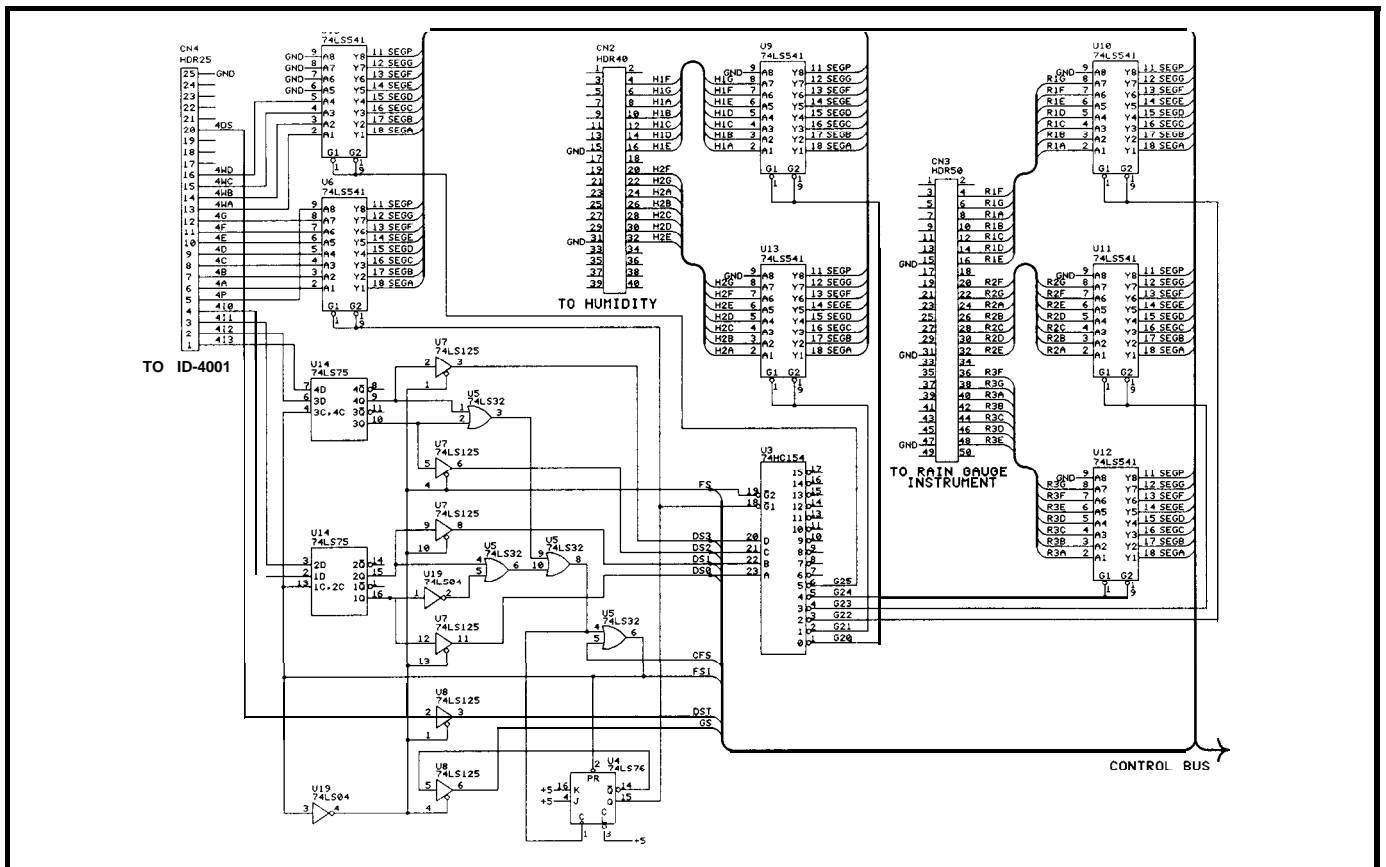


Figure 2 -- The Heath ID-4001 interface. (Note: This circuit could be used directly with a computer's parallel port.)

cated procedure for initial testing, I've built most of the functions into the firmware. Before booting the unit, however, remember to check the voltages and polarities on all IC sockets before installing the ICs, check the voltage on the MAX232 before installing the 68000, and install all jumpers (see Figure 1) before fully testing the unit. You should first see a title display, followed by the test sequence displays described in the last issue, followed by the "NO CONFIGURATION" message. If everything is well at this point, you've cleared the major hurdle and the basic hardware is functioning.

I'll devote much of the next installment to the PC Host software needed to configure the PP and download and process its instrument data.

### Instrument Interfacing

As mentioned in the past, I've initially provided the PP with the ability to handle data from the Heath ID-400 1 and ID-500 1 Weather Computers. The ID-5001 requires only the presence of a

serial port and the necessary firmware routines; the ID-4001 is a bit more complicated.

When Heath introduced the ID-4001, they provided a parallel port for moving the system's data to the outside world. Their intention was to interface the Weather Computer to another computer for more complete record keeping. However, my instincts say that the port probably was intended by the designer to be used as a remote display driver, and it would be very easy to use for this purpose. Using the port for complex computer interfacing is another story.

The computer port output data consists of the seven-segment representation of each of the ID-4001's front-panel display digits; the four digit-select lines determine which digit's data is on the output lines.

*[Editor's note: See "The Home Satellite Weather Center, Part 6" in Circuit Cellar INK #6 for more detail on the data from the Heathkit Weather Centers.]*

So far, so good. You also need

to monitor the four-bit (one-of-sixteen) wind direction data lines and use the strobes to validate the data.

You could let the PP take in all of this information and sort it as necessary, but it would be a massive waste of processor time to do so. Remember, you need to make sure that the precise and least controllable input (the WEFAX data) has the priority on processing time. You can't let the PP also be a slave to the ID-4001, catching nonsequential information "on the fly." In such a scenario, the PP would be responsible for capturing the data, converting it to ASCII, validating it (for instance, if a temperature changed from 99 degrees to 100 degrees during a sample, you might end up with either 199 degrees, or 00 degrees), and storing it in the proper order.

That much housekeeping would be more easily performed by a small hardware interface, so I've designed the device shown in Figure 2. The unit serves a number of purposes:

- It converts the seven-segment

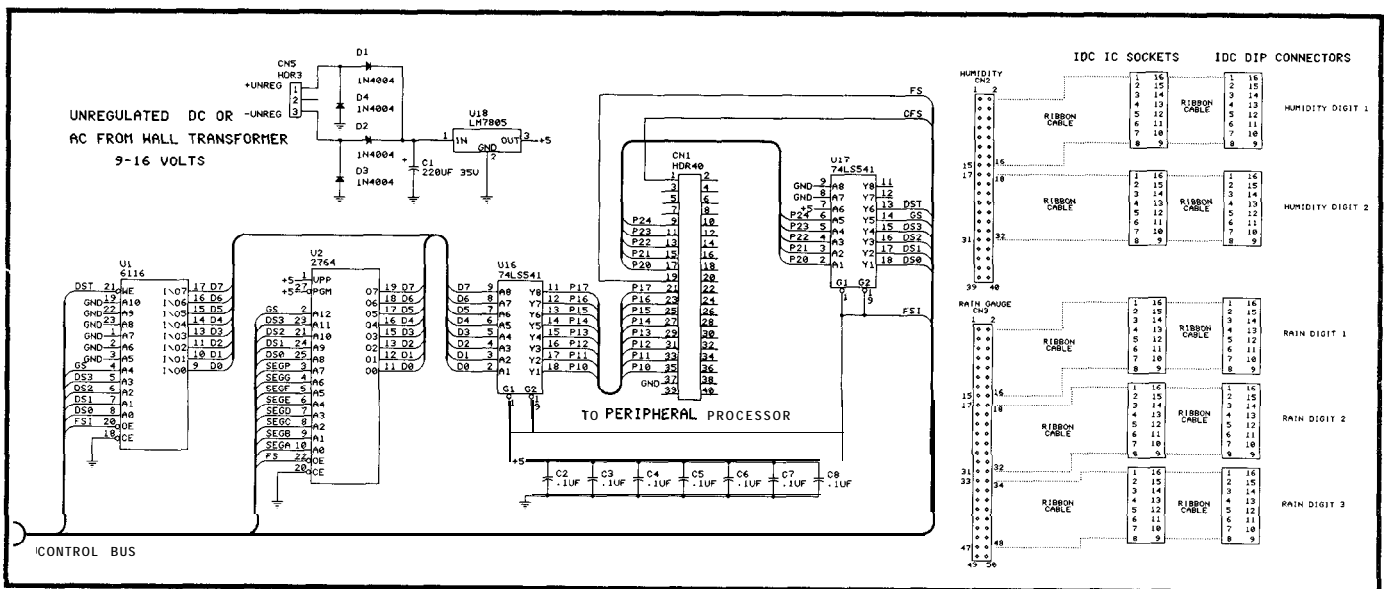


Figure 2 (continued) -- Cable assemblies for connection to rain gauge and humidity instrument. Remove decoder/driver/latch IC for each digit and plug in the corresponding DIP plug. Install the IC in the ribbon's socket.

data to ASCII.

- It stores the numerical data in a simple on-board SRAM for later transfer to the PP.

- It buffers the wind direction data for access by the PP.

- It provides for data conversion of the separate Digital Rain Gauge and the separate Humidity Indicator.

There's no real "black magic" in this project; a cable connects the ID-40013 parallel port to the interface, where (in the write mode) the seven-segment data is used with the strobe lines to address the ASCII-conversion EPROM. The data lines from the EPROM become the data lines to the SRAM, which is addressed by the digit-select lines. The data from the other instruments, if used, is strobed in on alternate digit-select

passes from the ID-4001 (A4 asserted disables the incoming data lines from the ID-4001, and polls the data latches involved with the extra instrument(s)).

The digital rain gauge and the humidity indicator were never intended to be interfaced to anything, so you'll have to be a little more clever with them. They do produce accessible data: the displays are driven by seven-segment drivers, so we have the same convertible data format as we have from the ID-4001. I take advantage of the segment drivers by using a custom-made ribbon cable consisting of two DIP plugs and a DIP socket. Remove the segment driver IC, plug the instrument end of the cable into the now-vacated socket, plug the IC into the cable's socket, and you have the data available at the interface end. I've designed the interface to minimize the load on the segment driver, which is really

only "monitored" anyway.

The read sequence is controlled by the PP's parallel port. The parallel port asserts its strobe line, causing the SRAM to be deselected from write mode on completion of the current digit-select cycle (digit select = 0, A4 negated). The parallel port then receives the acknowledgement through the input strobe line and begins a sequence of placing an address on the port output lines and reading the port input lines for the data until all addresses are read. The port output strobe is negated, and the write sequence resumes.

We'll provide power to this interface from the PP. The only other connections besides those to the parallel port would be to the ID-4001 and the other instruments via short lengths of ribbon cable.

That wraps things up for this issue. Next time, I'll discuss the PC Host software for communications

## Development Tools

**PseudoSam Cross-assemblers \$50.00**

**PseudoMax Cross-simulators \$100.00**

**PseudoSid Cross-disassemblers \$100.00**

**PseudoPack Developer's Package \$200.00 (\$50.00 Savings)**

### POWERFUL

PseudoCode is pleased to announce the release of an extensive line of professional cross-development tools. Tools that speed development of microprocessor based products. Fast, sophisticated macro assemblers to generate your program code. Versatile simulators that allow testing and debugging of the program even before the hardware exists. Easy to use disassemblers to help recover lost source programs.

### AFFORDABLE

Until now, powerful tools like these have been priced from 5 to 10 times! PseudoCode's price. Putting these time saving tools out of reach of all but large corporate engineering departments.

### BROAD RANGE OF SUPPORT

- PseudoCode currently has products for the following microprocessor families (with more in development):

Intel 8048

Motorola 6800

Hitachi 6301

Rockwell 65C02

Motorola 68000,8

RCA 1802,05

Motorola 6801

Motorola 6809

Intel 8080,85

Motorola 68010

Intel 8051

Motorola 6811

MOS Technology 6502

Zilog Z80, NSC 800

Intel 8096

Motorola 6805

WDC 65C02

Hitachi HD64180

- To place an order call one of our dealers:

Programmer's Connection USA (800) 336-166 INTL (216) 494-3781

KORE Inc. (616) 791-9333

PseudoCode

P.O. Box 1423

Newport News, VA 23601-0423

(804) 595-3703

## BCC52 BASIC-52 COMPUTER/CONTROLLER

The BCC52 Computer/Controller is Micromint's hottest selling standalone single-board microcomputer. Its cost-effective architecture needs only a power supply and terminal to become a complete development or end-use system, programmable in BASIC or machine language. The BCC52 uses Micromint's new 80C52-BASIC CMOS microprocessor which contains a ROM-resident 8K byte floating point BASIC-52 interpreter.

The BCC52 contains sockets for up to 48K bytes of RAM/EPROM, an "intelligent" 2764/126 EPROM programmer, 3 parallel ports, a serial terminal port with auto baud rate selection, a serial printer port, and it is bus compatible with the full line of BCC-bus expansion boards. The BCC52 bridges the gap between expensive programmable controllers and hard-to-justify price-sensitive control applications.

BASIC-52's full floating-point BASIC is fast and efficient enough for the most complicated tasks, while its cost-effective design allows it to be considered for many new areas of implementation. It can be used both for development and end-use applications.

Since the BASIC-52 is bus oriented, it supports the following Micromint expansion boards in any of Micromint's card cages with optional power supplies:

BCC22 Smart terminal board	BCC53 Memory and 6-port I/O exp. board
ADP500 User vocabulary, digitized speech board	BCC13 8-Channel 8-bit A/D converter
BCC25 LCD display board	BCC30 16-Channel 12-bit A/D converter
BCC33 3-port I/O expansion board	BCC18 Dual channel serial I/O board
BCC40D 8-Channel optoisolated I/O expansion board	BCC55 Prototyping board
BCC40R 8-Channel relay output board	BCC45 Stepper Motor board

**BCC52** BASIC-52 Controller board \$189.00

**BCC-SYST.5 '52 PAK\*** Starter System \$449.00

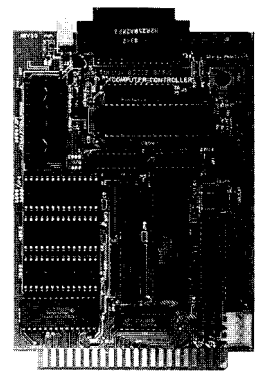
Includes: BCC52, ROM A&B, BUTIL., CC01, MB08, UPS10

**BCC52 OEM 100** Quantity Price \$149.00


**BCC52C** Lower power all-CMOS version \$199.00

Note: The BCC52 series is available in Industrial Temperature Range, fully tested. Prices start at \$294.00 single qty. Call for OEM pricing.

Micromint, Inc. — 4 Park Street, Vernon, CT 06066



To Order Call  
1-800-635-3355  
Tel: (203) 871-6170  
FAX: (203) 872-2204  
TELEX: 643331

and processing of the instrument data, cover any corrections needed on our past articles, and prepare to move into the area of WEFAX reception and processing. 

**IRS**

- 210 Very Useful
- 2 11 Moderately Useful
- 212 Not Useful

I am making kits available for each portion of the Home Weather Center system. Each kit, unless otherwise noted, consists of a PC board and all devices and parts (except SRAM or DRAM devices) to construct the standard design. Pricing for the kits is as follows (all include shipping charges):

68000 Main Processor Board..... \$319.00

Front-Panel Board..... \$88.00

1 Meg x 16 Memory Board.....\$189.00

Power Supply (quantities limited)..... \$40.00

Cabinet (quantities limited)..... \$44.00

I will be making individual components available for those with "well-stocked" parts cabinets. Send me a stamped, self-addressed business-size envelope for a complete listing, with prices.

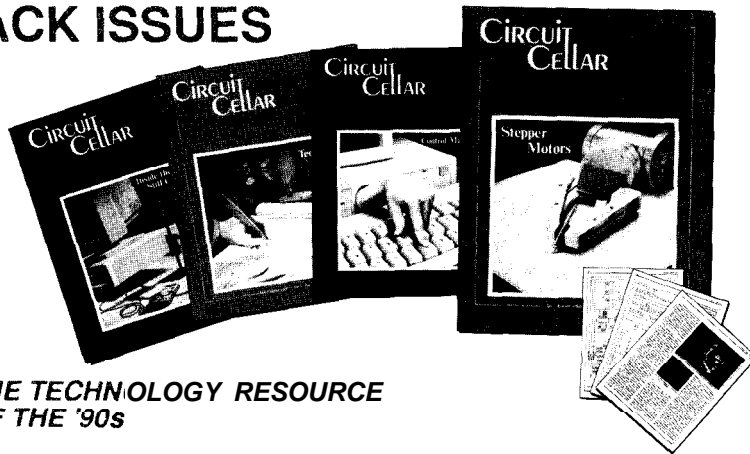
**Mail orders:**

**Mark Voorhees**  
**P.O. Box 27476**  
**Phoenix, AZ 85061-7476**

Include check or money order if ordering kits. I regret that, at this time, I am unable to accept credit cards.

Allow 4 to 6 weeks for shipment since most of this material will ship UPS.

**BACK ISSUES**



**THE TECHNOLOGY RESOURCE OF THE '90s**

**Issue #1**— *Inside the Box Still Counts* — **SOLD OUT**

**Issue #2** — *Techies* — **SOLD OUT**

**Issue #3** — *Control Magic* — **SOLD OUT**

**Issue #4** — *Stepper Motors* — **SOLD OUT**

**Issue #5** — *Remote Video Surveillance* — **GOING FAST**

- . ROVER: Remotely Operated Video-based Electronic Reconnaissance (Part 1)
- . The Home Satellite Weather Center: Focus on the MC68000 Peripheral Controller (Part 5)
- . IO-MHz&bit Digitizing Board for the IBM PC
- . Precision Pulses: Carrier Current Transmission Timing

**Issue #6** — *Data Acquisition*

- . ROVER: (Part 2) The Software
- . The Home Satellite Weather Center: Adding Serial and Parallel Ports to the Peripheral Controller (Part 6)
- . Build a Remote Analog Data Logger (Part 1)
- . ImageWise/PC — The Digitizing Continues (Part 1)
- . DDT-51 Revealed

Send \$4.00 per issue (includes shipping & handling) in check or money order to: Circuit Cellar INK, P.O. Box 772, Vernon, CT 06066. Visa and MasterCard accepted, call (203) 8752199

**CCINK's 1st Year Reprints**

Because so many of you save every issue of CCINK as a resource, our fast rise in circulation has resulted in a virtual sellout of the first year of INK. So as not to disappoint new subscribers as well as satisfy the demands of current readers, we are offering a B&W offset reprint of CCINK's first year (Issues 1-6). Available in February 1989, for \$20.00 in the U.S. and \$24.00 to Canada and Europe (shipping & handling included).

Send check or money order to:

Circuit Cellar INK  
**1st Year Reprint**  
 P.O. Box 772  
 Vernon, CT 06066

Visa and Mastercard will be accepted, call (203) 875-2199.

STATEMENT REQUIRED BY THE ACT OF AUGUST 12, 1970: SECTION 3685, TITLE 39, UNITED STATES CODE SHOWING THE OWNERSHIP MANAGEMENT AND CIRCULATION OF CIRCUIT CELLAR INK, published bi-monthly at 4 Park Street, Vernon, CT 06066. Annual subscription price is \$14.95. The names and addresses of the Publisher, Editorial Director, and Editor-in-Chief are: Publisher, Daniel J. Rodrigues, 4 Park St., Vernon, CT 06066, Editorial Director, Steven Ciarcia, 4 Park St., Vernon, CT 06066, Editor-in-Chief, Curtis Franklin, 4 Park St., Vernon, CT 06066. The owner is: Circuit Cellar Inc., Vernon, CT 06066. The names and addresses of stockholders holding one percent or more of the total amount of stock are: Steven Ciarcia, 4 Park St., Vernon, CT 06066. The average number of copies of each issue during the preceding 12 months are: A) Total number of copies printed: (net press run) 11,400 B) Paid Circulation: (1) Sales through dealers and carriers, street vendors and counter sales: 3,291 (2) Mail subscriptions: 5,962 C) Total paid circulation: 9,153 D) Free distribution by mail, carrier or other means: samples, complimentary, and other free copies: 450 E) Total Distributed: 9,603 F) Copies not distributed: 1) Office use, left over, unaccounted, spoiled after printing: 1,617 2) Returns from News Agents: 150 G) Total: 11,400 actual number of copies of the single issue published nearest to filing date are: (#5 Sept/Oct A) Total number of copies printed: (net press run) 22,000 B) Paid Circulation: (1) Sales through dealers and carriers, street vendors and counter sales: 7,900 (2) Mail subscriptions: 9,613 C) Total paid circulation: 17,513 D) Free distribution by mail, carrier or other means: samples, complimentary, and other free copies: 1,800 E) Total Distributed: 19,313 F) Copies not distributed: 1) Office use, left over, unaccounted, spoiled after printing: 1,980 2) Returns for News Agents: 707 G) Total: 22,000 I certify that the statements made by me above are correct and complete: Daniel J. Rodrigues - Publisher

# FROM THE BENCH

## AC Power Line Transmission

Conducted by Jeff Bachiochi

The power line network of copper and aluminum in this country delivers life-giving sustenance to all our inanimate appliances. However, it seems the only time we are conscious of this fact is when we pay the utility bill and examine the relationship between the rising kilowatt-hour demand and the amount due. Whether we consider our appliances as necessities is not the issue here. What can be clearly seen is the medium already in place which connects any number of points together (as long as there is an AC outlet within reach).

There are at least four problems when dealing with using the AC power line for data transmission: power line impedance, attenuation, impulse noise, and continuous interference, each playing havoc with attempts to use the power line network for anything other than an electrical lifeline.

Line impedance varies continuously, depending on the equipment on the line. Every time an appliance

turns on or off, by either automatic (e.g., refrigerator) or manual (e.g., hair dryer) control, the impedance is affected. Line impedance can range from below 1 ohm to above 50 ohms.

A 125-kHz signal will be attenuated about 7 dB (1/5 of its original power) for each 50 meters of cable. An additional 40 dB of attenuation (1 / 10000 of initial strength) will occur between phases of the power line. Any floors (in a factory or commercial complex) or houses (in a residential area) on the same side of a distribution transformer may receive the signal (though heavily attenuated by distance). Heavy losses through transformers designed for optimum coupling at 60 Hz reduce the carrier frequency to undetectable levels.

Line transients and glitches caused by changing loads can sometimes reach a few thousand volts. This very-short-duration impulse noise passes easily through any 60-Hz filtering.

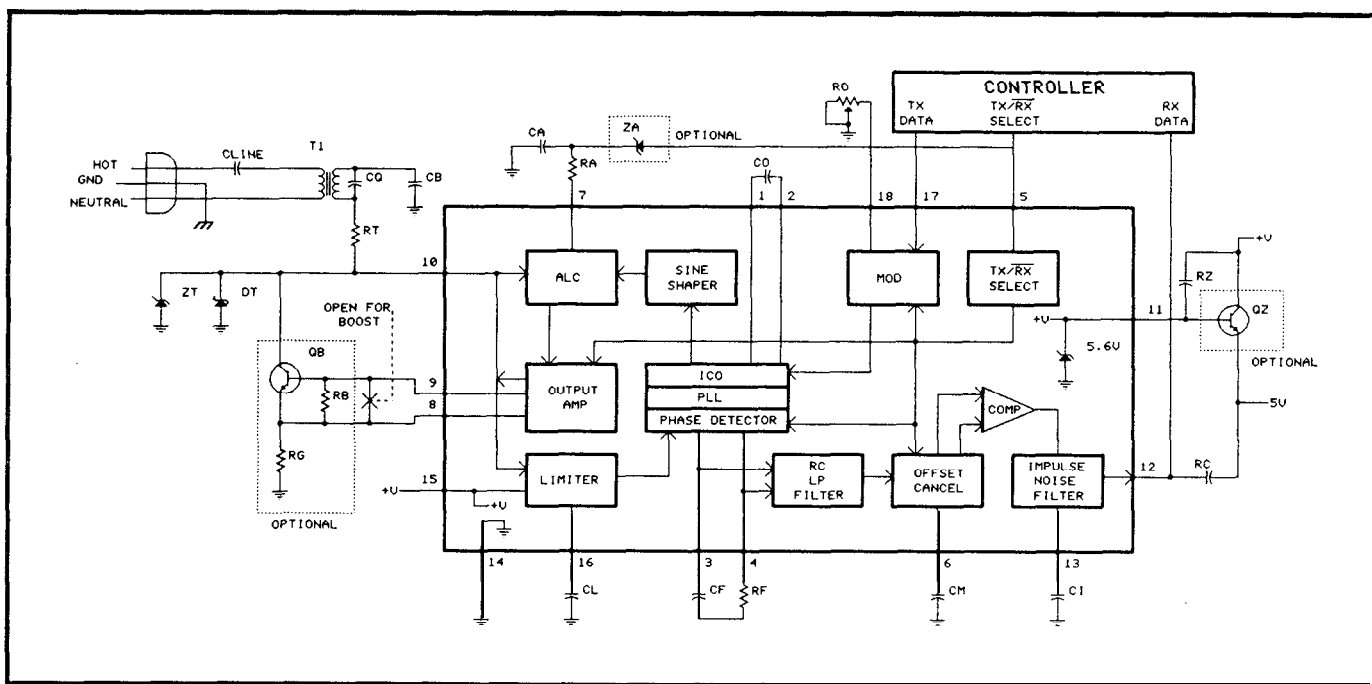
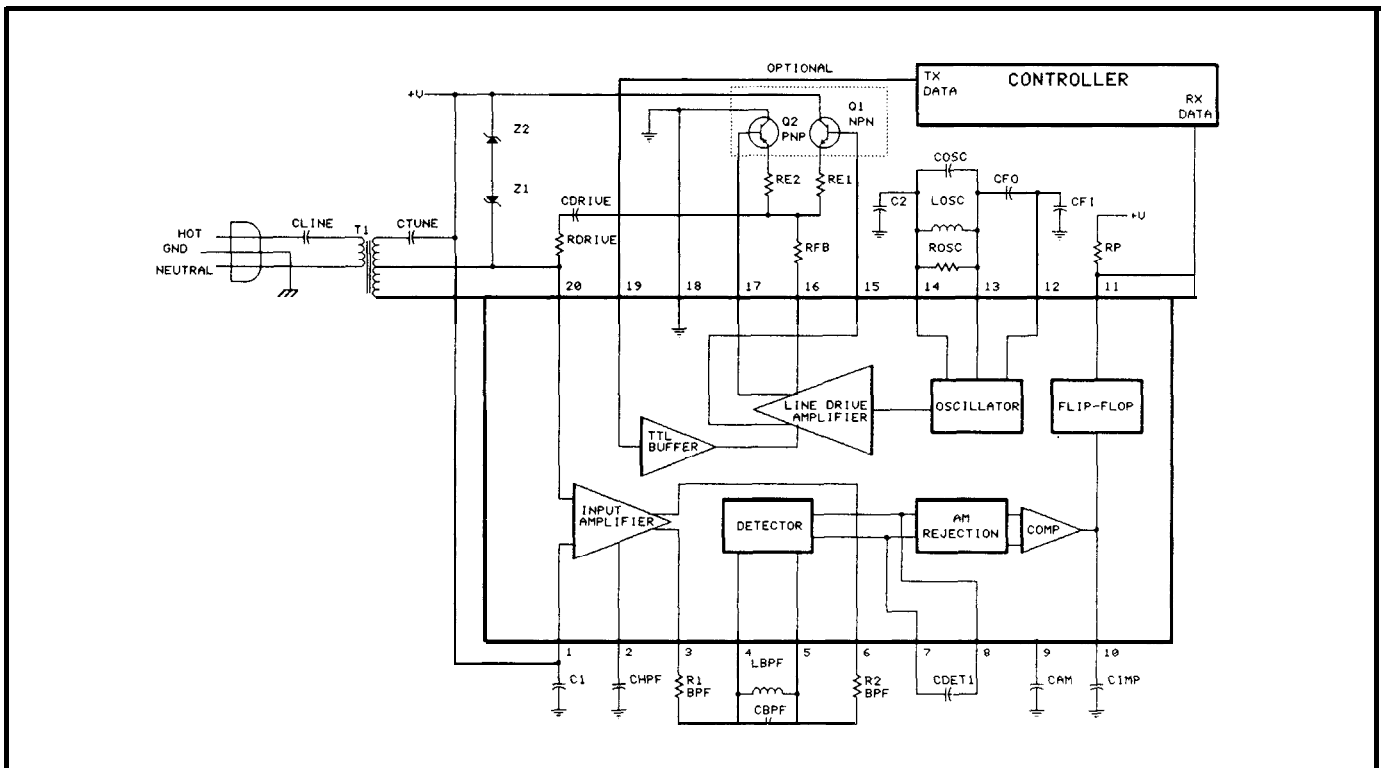


Figure 1 -- Stock diagram of the National LM1893 with surrounding support circuitry.



**Figure 2 -- The Signetics NE5050 needs a minimum of external components to support power-line-based communication.**

Continuous wave (CW) interference is caused by equipment generating a signal at some periodic frequency on the line. For example, high-intensity discharge (HID) lamps often use electronic ballasts which operate at high frequencies and can generate spectral harmonics. CW interference within the range of the selected carrier frequency will affect both the reception of the data and the quality of the data received.

The most important consideration for Power Line Modem (PLM) design is the carrier frequency. Close inspection of the AC line environment is required to reveal any CW interference, and a carrier frequency must be chosen in the range of least interference. The characteristics of driver chips limit you to a frequency somewhere between 50 and 300 kHz. (X-10 devices, which permit remote power on/off control of lights and appliances through the power line, use a carrier frequency of 120 kHz.)

#### National LM1893

National's BI-LINE LM1893, shown in Figure 1, is packaged in an 18-pin IC (the LM2893, a 20-pin version, has separate carrier in and out).

Transmission is started by selecting the TX mode and sending TTL serial data into pin 17. The data generates switched control current (MOD) to drive the

current-controlled oscillator (ICO) which modulates to  $\pm 2\%$  of the carrier frequency. A sine shaper removes most of the unwanted harmonic content of the signal, which would cause unwanted RF overtones. Automatic level control (ALC) shunts excessive drive current away from the output amplifier to keep the signal from clipping due to line impedance changes which would cause unwanted RFI. Finally, the carrier is placed on the AC line through a coupling transformer. The LM1893's output drive current can be boosted by using a single external transistor.

When the RX mode is selected, the LM1893 becomes a receiver. Signals passed through the coupling transformer are input to the CARRIER I/O pin. A balanced Norton-input amplifier (LIMITER) removes the DC offsets, attenuates the line frequency, acts as a band-pass filter, and limits the drive signal to the PLL phase detector. The differential demodulated output contains AC and DC data, noise, system DC offsets, and a 2x carrier frequency component. A three-stage RC low-pass filter removes most of the carrier components while the OFFSET CANCEL compensates for offsets by adjusting a series correction voltage. Data and impulse noise from the comparator drive the impulse noise filter integrator capacitor. All of the impulse noise less than the integrator time constant is removed and the data is passed to the DATA OUT pin.

## Signetics NE5050

The Signetics **NE5050**, shown in Figure 2, is packaged in a 20-pin chip.

The Signetics NE5050 does not have a TX/RX mode. It will receive its own transmissions, which can be a useful diagnostic tool. FSK modulation requires two **NE5050s** per node, while ASK modulation (amplitude shift-keying) needs but one. The example circuit shown in Figure 2 uses ASK modulation. TTL data into the TX IN pin gates the carrier frequency oscillator through the line-drive amplifier, and the carrier is placed on the AC line through a coupling transformer. The NE5050 can use an optional external complementary transistor pair for increased current capability.

The receiver of the NE5050 is always active. Signals passed through the coupling transformer are input to the RX IN pin. A balanced Norton-input amplifier removes the DC offsets, attenuates the line frequency, acts as a band-pass filter, and limits the differential drive to the external interstage band-pass filter. This external circuit can be as simple as a passive RCL network or as elaborate as an active filter arrangement using ceramic filters. The externally filtered signal reenters the NE5050 at pins 4 and 5. A Gilbert detector compares the in-phase signals, and outputs a demodulated fully rectified signal across a differential capacitive load. The AM rejector tracks the DC value of the signal, adjusting a series voltage to that of the detector capacitor. The comparator supplies a constant current to the impulse capacitor, charging and discharging it at a constant slope. The narrow impulse noise will not last long enough to fully charge or discharge the capacitor. Two volts of hysteresis ensures that the impulse noise will not affect the SR flip-flop. Only data will toggle the open-collector output, RX OUT, at pin 11.


With both the LM1893 and the NE5050, tradeoffs must be made between data rate and noise immunity. Data rates greater than or equal to 4800 bps may not be acceptable in harsh environments, so slower rates may have to be used.

### A Delicate Balance

Due to the relationship between CW interference and carrier frequency, and also between impulse noise and maximum data rate, component values should be selected for an individual environment and will not be discussed here. Each manufacturer fully covers component selection and typical application examples in their respective data books.

A word on protocol. Any node can access the line to transmit signals at any time. This can cause data

collision, and can result in loss or corruption of data, so some intelligence is necessary to prevent two transmitters from colliding. Refer to articles on Ethernet, XMODEM, and other protocols for more information. Since data bit errors are related to noise immunity, the number of bit errors will go up as the data rate goes up; error checking is an absolute necessity at higher data rates. Most telecommunication packages can send files with some type of error checking or error correcting protocol, but the communication channel itself doesn't have any error checking, so data sent without a protocol is susceptible to line noise.

Whether or not PLM communication is practical for you will depend on a number of factors. How much information is to be sent? How fast? What is the line environment? Do costs compare favorably to alternative systems? 

### References

Linear Data Manual Volume 1: Communications  
Signetics Corporation  
811 East Arques Ave.  
P.O. Box 3409  
Sunnyvale, CA 94088-3409  
(408) 99 I-2000

Linear Databook 3  
National Semiconductor Corporation  
2900 Semiconductor Dr.  
P.O. Box 58090  
Santa Clara, CA 95052-8090  
(408) 721-5000

#### WARNING: ELECTRICAL SHOCK HAZARD!

In addition to supplying 110 VAC, the power line is a near-infinite source of current. As little as 100 mA is sufficient to kill a human being. Exercise extreme caution when using either device described here. Any capacitors wired on the per line side of the circuit can retain a charge even after power has been removed from the circuit.

Innovations like these help to make today's technology more cost effective, reliable, and easier to use. Please share your favorite ideas, chips, and circuits with others.

We will pay \$25 for any *From the Bench* accepted for publication. **All** submissions should be typed, double-spaced, and include neatly drawn schematics or Schema configuration, library, and page files.

Include a stamped, self-addressed envelope large enough to hold everything if you wish the materials that have not been accepted to be returned.

Submit to:

From the Bench  
c/o **Circuit Cellar** INK  
P.O. Box 772  
Vernon, CT 06066

IRC

213 Very Useful  
214 Moderately Useful  
215 Not Useful



# Writing a Real-Time Operating System

## Part 1

by Jack Ganssle

### *A Multitasking Event Scheduler for the HD64180*

Mention the phrase "Real-Time Operating System" at a software convention and you'll evoke as many different reactions as there are programmers. VAX programmers may think of VMS, PDP-11 sysops will begin discoursing on RSX-1 IM, but many microprocessor designers will respond with a blank stare. The Real-Time Operating System (RTOS), though long the backbone of mainframe and mini systems, is only now coming into its own as a component of embedded systems.

An RTOS is quite different from MS-DOS, the most recognizable of operating systems. True, MS-DOS does provide a file structure and device controllers (which invariably seem to be bypassed in most real applications), so it does qualify as an OS. However, MS-DOS can handle only a single program at any time; it is incapable of multitasking.

Another class of RTOS has evolved specifically to support embedded systems (those that don't rely on disks, such as a race car controller or a microwave oven). The "Embedded Real-Time Operating System" provides multitasking resources for ROM-based applications. It differs from a mainframe operating system in that it offers no disk support or file structure.

Why use an RTOS?

Most embedded applications

process numerous asynchronous external events. For example, a microwave oven controller must scan the keypad, control the magnetron, generate a countdown timer display, and handle the various door switches and other safety devices. While each activity is simple, overall coordination is not.

An inexperienced programmer might try to write the microwave's code in a sequential manner -- read the keypad, perform the requested function, reread the keypad, and so on. What initially looked simple soon turns into a nightmarish mix of spaghetti code where all routines are inexorably intertwined. Suppose the door is opened while the oven is on? Have the magnetron routine check the door switch. Did the galloping gourmet hit the "Cancel" button? Add a keyboard scan routine to the magnetron code. If the CPU is tied up with the magnetron, how do we count down and display time? Add yet more code to the magnetron subroutine.

This apparently simple application becomes very complicated, violating The First Fundamental Rule of Programming: it's OK for a program to be complex, but no individual subroutine may be. If a subroutine seems to be getting out of hand, step back, think hard, and take another approach.

In this case only a Real-Time Operating System will clean up the code (you just knew that was coming!). Each logically independent activity becomes a "task." Every

task runs asynchronously, competing for CPU time. Like Quantum Mechanics, there is no way to determine exactly when each task will run, and there is even less need to.

In the case of the microwave oven, an RTOS serves as the software's backbone. A magnetron handler is one simple task. A keypad driver is another. Other tasks handle the switches, timer, display, and so on. The RTOS controls the overall sequencing of each of the tasks. For example, the RTOS can run the display driver once every 200 milliseconds -- fast enough to cause no noticeable delay, yet slow enough to not burden the processor unnecessarily. The magnetron routine can run once a second, the typical timing resolution offered to the cook. The keypad task should run rather frequently so it can **debounce** the buttons properly.

This makes the sequencing almost trivial. The door switch task will issue a command to the operating system to cancel the magnetron task if the door is opened -- regardless of whether microwaves are being generated. Similarly, the keypad task will cancel the same activity if the "abort" button is pressed. Since the system is (hopefully) designed in a fail-safe manner, the magnetron task will only have to pulse a one-shot line to the control circuits. It no longer has to make decisions.

All real-time controllers face similar multitasking requirements.

In recent years several vendors have introduced commercial **Real-Time Operating Systems** to satisfy this growing need. Intel's **iRMX** is certainly the best known. Avocet's **AVRX51** is designed for 8051-series single-chip computers. Although the commercial products are all excellent, they have taken the fun out of multitasking! To me, the RTOS is the most interesting part of the software.

The **BCC180** is the ideal platform for an RTOS. Its processor includes the timer needed to sequence tasking. The built-in memory manager lets it support huge programs. A tremendous amount of memory (for an 8-bit system) is included on the board.

The whole premise of a multitasking operating system is that computer time is a valuable resource to be hoarded and allocated wisely. This precludes code that polls devices, waiting for an event to occur. Rather, devices use interrupts to signal their readiness to accept or return data. The burden of letting software know that the device is ready is thus shifted to the hardware, where it belongs.

Further, at times any individual task may need to go idle. Rather than execute a null loop, the task should signal the RTOS its need to suspend execution for a specific time. To the RTOS, this bonus CPU time is a bonanza that can be allocated to other tasks.

These requirements define the central nucleus of the RTOS. To provide proper task execution, the RTOS must:

-- Switch execution time between tasks, so every task gets its fair share of time. This is called time slicing.

-- Provide a mechanism whereby tasks can go into an idle state for specified periods, without wasting CPU time. In the RTOS to be described, **WAITing** and scheduling are the methods used.

-- Allow tasks to alter their execution order when some important event occurs. This involves a particular task altering its "priority" level.

With these requirements in mind we're now ready to examine a real RTOS.

## The **BCC180** RTOS

The **BCC180's** Real-Time Operating System implements all of the previously described functions. The operating system itself consists of two major segments: the context switcher, which is responsible for starting, stopping, and sequencing tasks; and the task control code, which is used to initiate various task operations.

The RTOS itself resides in low memory as a separate section of code. The context switcher cannot be accessed by the user's code; it is invoked only on an interrupt from the timer. A number of subroutines can be called by application programs to request task servicing.

**BCC180** RTOS uses the **HD64180's** timer 0 to sequence all task activities. It is programmed by the initialization routine to generate an interrupt every 10 milliseconds. Every time this interrupt occurs, the currently executing task is suspended and the context switcher invoked.

This regular source of interrupts forms the heartbeat of the operating system. It's the mechanism by which the CPU can be shared between many activities. The whole philosophy is to allocate CPU time to tasks in IO-*ms* chunks.

## Task States

Any task can be in one of five states:

**DORMANT** -- the task has no need for CPU time.

**READY** -- the task will require the CPU when its "reschedule time" (defined later) has elapsed. It

is not immediately a contender for execution time.

**ACTIVE** -- the task is executing. Obviously, only one task can be active at any time.

**WAITING** -- the task has requested a delay. After the delay time is up the task again becomes available for execution.

**SUSPENDED** -- all conditions are satisfied for the task to run, and the task was at one time running. It was "suspended" by an interrupt from the timer tic. The task is anxiously awaiting CPU time from the context switcher.

Figure 1 is **RTOS's** state diagram. The circles represent each possible task state. The arrows show the event causing a transition between states. Since many tasks are competing for time, Figure 1 is somewhat of a simplification of reality; in effect this should be a three-dimensional drawing, with "depth" added for each task. Since only one task can be active at any time, there will be only one **ACTIVE** circle for the entire picture; **ACTIVE** serves as the hub around which all task sequencing flows.

Tasks are initially defined using the **OS\_DEFINE** subroutine. **OS\_DEFINE** makes an entry in the task control block (TCB) for the task, so the RTOS knows about it. Tasks are initially **DORMANT**. The context switcher is aware of **DORMANT** tasks, but ignores them until an **ACTIVE** task commands a **DORMANT** one to go to the **READY** state. This can only happen as a result of the **ACTIVE** task issuing a call to the operating system's **OS\_RUN** subroutine. When a task enters the **READY** state it is not immediately eligible for execution (i.e., it is not allowed to go directly to **ACTIVE**). **RUN** is called with a reschedule interval for the task. This task may not start until the number of tics specified in the **RSI** parameter elapses.

Many operating systems don't completely exploit the possibilities

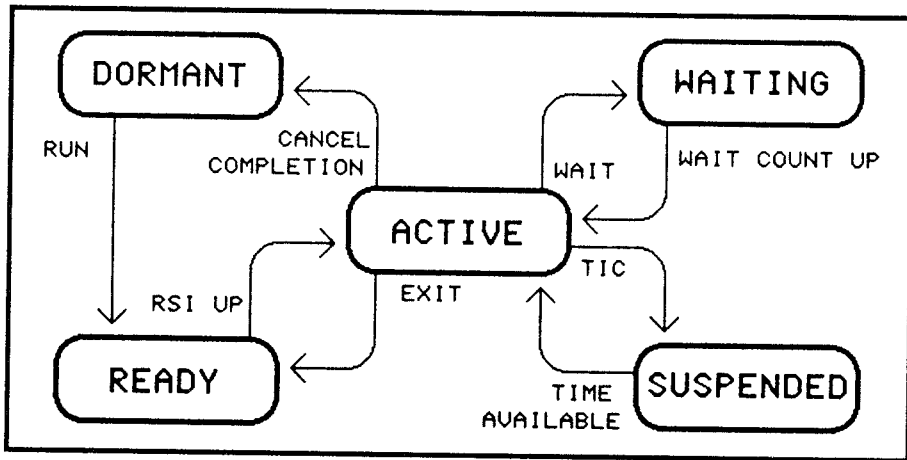


Figure 1 -- The state diagram for our real-time operating system clearly shows its operational states and the transitions from one state to another.

of multitasking. They allow tasks to be created, executed, and reexecuted at any time. However, the tasks often can be restarted only "manually"; some other task must issue a call to the RTOS to restart a completed task. In many cases, however, it makes more sense to allow the operating system to automatically restart completed tasks at regular intervals. This makes it possible to implement a real-time clock outside of the OS proper. An example is an LCD driver that must update a display, say, once every 200 ms. The task that updates the display can do its work very quickly and exit; the RTOS will restart it every 200 ms.

The BCC180 RTOS supports automatic task restart through a concept called "rescheduling." Whenever a task is started, a "Reschedule Interval," or RSI, is specified. While the task is executing, the RSI is ignored. Once the task runs to completion, however, the RTOS counts down time until the RSI elapses, and then restarts the task. The best analogy is to reincarnation, except the task is always reborn as software, and not as a higher animal.

When a READY task's RSI is up, the task is finally eligible for execution. Subject to demands made by other competing tasks, the operating system will raise the task to ACTIVE and start executing it as

time permits.

Very short tasks may complete before another tic comes (10 ms -- many thousands of instructions). If this happens, the task is placed back into the READY state, where it remains until its reschedule interval once again elapses. This process repeats forever, unless the application program commands a change via the CANCEL or WAITING calls.

Many tasks will take more than one tic's worth of time to complete. Indeed, some may run forever (if they include an infinite loop -- perfectly legal and valid in a multitasking system). What is the RTOS to do? Remember that the *raison d'être* for a real-time operating system is to wisely allocate limited processor resources to many competing tasks. Although each task may want 100% of the processor's time, it simply won't get it.

The RTOS has only one option: put the ACTIVE task in a SUSPENDED state and give another a chance to run. SUSPENDED means exactly what it implies. The task needs more computer time, but has been put on hold. The context switcher must very carefully preserve the complete state of the task, including all of its registers, stack, flags, etc., so the task can be resumed without ever being aware it had been interrupted.

A READY task makes no de-

mands for CPU time (it is waiting for the RSI to elapse), a DORMANT task practically doesn't exist, and, as we'll see, a WAITING task is also in limbo. Only those SUSPENDED are engaged in a desperate competition for processor time. You can see them in the distance -- a long line of war-weary suspended tasks, their stack pointers askew, tense with anticipation, each waiting to be called to the front . . .

Sometimes a task must go idle until an external event occurs. One approach is to EXIT, going back to the READY state until the RSI goes by. The task will be completely restarted when it goes ACTIVE. It will run "from the top," which may not be desirable. We need a way to place the task in suspended animation for a time. Obviously a null loop will do just this, but at the expense of wasting CPU time.

The WAITING state is provided to allow a task to delay for a fixed time, and to restart from the point where the delay was initiated. The application program initiates a WAIT by issuing a call to OS\_WAIT, requesting the ACTIVE task be placed in a WAITING state for a specified number of tics. The context switcher will not give the task access to the CPU until the wait count elapses.

Finally, any task can return to the DORMANT state, taking it

## MORE GOOD CODE... FAST!

Softaid's In-Circuit Emulators give you all the power and speed you need to develop microprocessor based products in realtime increasing your productivity and saving you time and money.



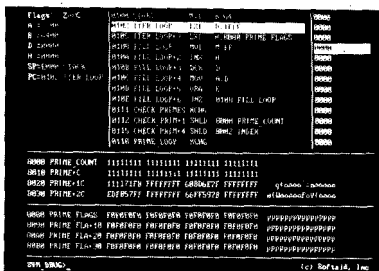
Emulators available for:

64180, Z80, Z180, 8088/18086  
80188/80186, 8085, V40/V50

Priced from \$595 to \$2995

## FULL SCREEN DEBUGGING!

With the optional source level debugger, you get a real time, full screen debugging environment with pop-up windows and symbolic displays. Your source code and comments are displayed in a window that is automatically linked to the debugging session. This makes embedded system debugging FAST and EASY!



## TIMELY TECHNICAL SUPPORT!

Our technical staff is ready to answer your questions. Give us a call to discuss your microprocessor development needs!

Complete information is also available on our BBS from 5 p.m. to 3 a.m. EST -- 301-964-8456.



8930 ROUTE 108  
COLUMBIA, MD 21045  
(301) 964-8455  
(800) 433-8812

completely out of contention for computer time. Since a task doesn't truly die by running to completion (it will be reincarnated after the RSI elapses), another mechanism is needed to completely remove it from the execution stream. The OS-CANCEL subroutine tells the RTOS a specified task is to lose its rescheduling privileges (by driving the task to DORMANT).

OS-CANCEL doesn't abort the task; rather, it signals the operating system that the task should be allowed to complete and then never be reincarnated.

Once canceled, a task can be restarted by calling OS\_RUN again. When OS\_RUN is called, the entire rescheduling path will recommence.

## The Scheduling Algorithm

The previous description shows the transitions between task states, but exactly how does the context switcher decide which of the many competing tasks to elevate to ACTIVE and start running? After all, when a tick is received and the context switcher invoked, dozens of tasks could be suspended, WAITING (with the wait count elapsed), and READY (also with RSI up).

If more than one task is eligible for execution, the context switcher must pick one to run in such a manner as to satisfy two conditions:

- 1) Every task must get a fair chance to run.
- 2) Certain crucial tasks are more eligible than others, and must be given more opportunity to execute.

To ensure that every task gets the same chance to run, the RTOS maintains a pointer (TCBPTR) to the last task it executed. When a tick invokes the context switcher, it attempts to run the next sequential task. If that task is not eligible for

## CIRCUIT CELLAR INK'S ADVERTISER'S INDEX

Reader Service Number		Page Number
101	AISI Research	C4
102	Alpha Products	7
.	Assoc. Comp Consult.	63
103	AVOCET	c2
*	Best Associates	63
104	Binary Technologies	58
105	Cabbage Cases	33
*	Chrysalis Micro.	63
106	Circuit Cellar	35
107	Circuit Cellar	19
.	Collins Associates	63
106	Cottage Resources	33
109	Covox, Inc.	25
*	David Baker Assoc.	63
110	Environmental Optics	30
111	Galacticomm	Outsert
112	Hogware	35
113	Innotec Design	24
114	Introl Corp.	31
115/116	JDR Microsystems	23
117	LTS/C Corp.	27
118	Micromint, Inc.	40
119	Micromint, Inc.	61
120	Micromint, Inc.	C3
121	PseudoCode	40
122	Schnedler Systems	25
123	SoftAid, Inc.	46
124	Thinking Tools	6
125	Timeline, Inc.	4
*	Tinney	33
126	x-10	15

## IRS INK Rating Service

How useful is this article?

At the end of each article and some features there are three 3-digit numbers by which you can rate the article or feature.

Please take the time to let us, at Circuit Cellar INK, know how you feel our material rates with you. Just circle the numbers on the attached card.

<Circle No. 123 on Reader Service Card

execution, then the operating system continues searching for one to run. In other words, it tries to run the one that has not been ACTIVE for the longest amount of time. This is called Round Robin scheduling, and is the basis for all operating systems.

The Round Robin scheduling algorithm always guarantees that every task gets an equal chance to run, but suppose the application program defines a task so important that when it is ready to go ACTIVE it absolutely must gain control of the processor? For example, if a task were keeps track of time, it MUST execute every so many tics, or the clock will get behind.

BCC180 RTOS provides a way to associate a priority with each task. The higher the priority, the more important the task is. Priorities can range from 1 to 63, which are relative, **unitless** numbers. When a task is defined with the OS-DEFINE routine, an initial priority is assigned to it. Later, the task can, at any time, alter its priority level by issuing a call to OS\_PRIORITY. Generally, all tasks should be assigned a **midlevel** (say, 32) priority, so others can raise their level above the mean, and some can lower theirs.

Why would you want a task to have low priority? In a real-time system, data acquisition is usually the most important activity since the data is available for a limited amount of time. Many tasks may be needed to gather data, analyze it, and then actuate a controller (perhaps for a closed-loop system). Often a display is also needed. The display task might run at an extremely low priority. The real meat of the application will be unaffected. The display activity may get suspended for long periods, but these delays may not be noticeable to the operator. Suppose a 200-ms update **rate** occasionally becomes 250 ms? No one will notice.

**BCC180** RTOS also lets tasks

Byte 0	Task state
1	1 if a cancel was requested for this task. When the task exits, if this byte is set the task becomes ineligible for further rescheduling.
2	Task's bank. This is the BBR value for the-task. Required to support memory management (described later)
3	Task's priority. Legal values are 1 to 63.
4	Task's reschedule interval. Given when the task is first requested for execution.
6	Task's reschedule count. This is set to the RSI every time the task exits, so the context switcher can begin counting down again.
8	Wait count. If the task is WAITING, this count is decremented to 0, at which time it again becomes eligible for execution.
10	Task's Start Address. This is the task's entry point. When a READY task is elevated to ACTIVE. it starts at this address.
12	Task's stack pointer. The current value of the SP is saved for all tasks so the context switcher can completely restore the task's state.

**Table 1 -- The Task Control Block (TCB) is the central data structure for the operating system. All scheduling information and handling of requests must go through the TCB.**

dynamically raise and lower their priorities through calls to OS\_PRIORITY. Obviously, a high-priority task is a dangerous beast, since it can easily hog all of the processor's time. Careful design of the application is essential so that CPU time becomes available to all tasks. It is much better to design a task to go to a high priority when it really needs to, and then to return to a normal level, than to keep it elevated at all times.

The priority scheme, then, alters the round robin algorithm. Instead of just starting the next eligible task, the context switcher actually first examines the current priority level of all tasks that are READY (with their RSI up), WAITING (with the wait count elapsed), and SUSPENDED. It starts the highest-priority task it finds that satisfies these conditions.

If several tasks are found that are eligible and that have equally high priority, the context switcher resumes the round robin concept. It alternates execution between the high-priority tasks until they all become ineligible. The next priority level is then executed. Obviously, this algorithm defaults to simple round robin if all tasks are at equal priority.

Remember: high-priority tasks

can completely monopolize the CPU! Be sure that they exit, wait, or get canceled fairly often so other tasks get a shot at running.

### The Task Control Block

Programs have been accurately defined as algorithms plus data structures. The algorithms have been described. These all revolve around one data structure, the Task Control Block (TCB). All scheduling information is kept in the TCB; all requests for service are made through it.

The TCB consists of one 16-byte entry for each task. Only 14 bytes are used, leaving two for future expansion. Its format is shown in Table 1. The TCBPTR pointer is always kept pointing to the currently ACTIVE task. It is used to implement the round robin algorithm.

The first TCB entry is assigned to "task 0," the main routine that must exist just to spawn other tasks. Task 0 is never explicitly created; the RTOS forces it to exist when a call to the operating system's initialize routine is executed.

The last TCB entry is the task's current stack pointer. Every task MUST use its own, distinct stack. If one stack were shared between

many tasks, after several tics the stack would become a horrible jumble with no way to accurately pair values and tasks.

When a tic interrupts a task, the context switcher pushes the entire state of the machine (all registers and flags) onto the current stack (i.e., that which belongs to the task just interrupted). The stack pointer is then saved in the TCB. When another task is started, that task's stack pointer is recovered from its saved position in the TCB, and used. POPs in the exact reverse order balance the stack and recover the task's register set. In this way every task's registers, flags, and stack are preserved, guaranteeing the integrity of the task's operation. In effect, the task never knows it was interrupted.

This has an important implication: each task MUST define an initial stack pointer via OS-DEFINE. Further, the stacks must be in logically distinct areas.

### The Code

With the RTOS completely described, the code becomes almost trivial. This is usually the case with software. After the programmer completely understands a problem and has the algorithms in mind, coding becomes almost a boring, low-IQ task (pardon the pun). And so it should be. Most software disasters are attributable to insufficient understanding of the problem, and not enough planning of its implementation.

Like Gaul, the code is divided into three sections. The RTOS proper resides in file OS.MAC. A number of macros designed to make the application code more readable are in OSMACRO.MAC. Finally, the user's application is in one or more files. The example application is called OSAPP.MAC.

**[Editor's Note: Complete code for this article is available from the**

```

context-switch:
    push    hl                ; push all registers
    push    de
    push    bc
    push    af
    push    ix
    push    iy
    exx
    push    hl
    push    de
    push    bc
    exx
    ex      af,af'
    push   af
    ex     af,af'
    in0    tcr                ; read tcr to clear interrupt
    in0    tmr01             ; also must read tmr to clear intr
    ld     ix,(tcbptr)       ; pt to current task's tcb entry
    ld     (ix+t_state),suspend; suspend the task
cs:
    ld     hl,0
    add    hl,sp              ; hl=stack pointer for this task
    ld     (ix+t_sp),l        ; save sp low in tcb
    ld     (ix+t_sp+1),h     ; save sp high in tcb
    call   dec_cnts          ; decrement tcb counts
    call   inc_rr_ptr        ; increment the round robin pointer
    call   find_tsk         ; find a task to execute
    ld     (tcbptr),ix       ; reset tcbptr
    ld     l,(ix+t_sp)       ; get low sp
    ld     h,(ix+t_sp+1)     ; get high sp
    ld     sp,hl             ; set the task's sp
    ld     a,(ix+t_state)    ; ix pts to task to run; get state
    ld     (ix+t_state),active; set task will now be active
    cp     ready             ; possible states: ready, suspended,waiting
    jr     nz,css1          ; j if not ready
    ld     l,(ix+t_start)    ; get low start address
    ld     h,(ix+t_start+1); get high start address
    call   remap             ; set proper bank
    push   hl                ; push start address
    ei
    reti                                ; start task
css1:
    cp     suspend          ; suspended?
    jr     nz,csw1         ; j if not; must be waiting
    call   remap           ; set bank for this task
    ex     af,af'         ; restore registers
    pop    af
    ex     af,af'
    exx
    pop    bc
    pop    de
    pop    hl
    ei
    pop    iy
    pop    ix
    pop    af
    pop    bc
    pop    de
    pop    hl
    ei
    reti                                ; resume suspended task
csw1:
    call   remap           ; remap to resume task that was waiting
    ei
    reti                                ; restart task

```

Listing 1

**Circuit Cellar BBS or on Circuit Cellar INK Software On Disk #7. For ordering and downloading information, see page 62.]**

A short initialization routine, OS\_INIT, must be called before the application issues any other RTOS calls. OS\_INIT sets up the TCB so that all subsequent calls to other routines will find the TCB in a known, safe state. This involves setting all tasks to a default

“NONE,” or nonexistent, state except for task 0. Task 0 is defined automatically as the application code that spawns off other tasks. OS\_INIT also programs the processor's timer 0 to interrupt every 10 ms.

CONTEXTSWITCH, shown in Listing 1, is the heart of the operating system. It can never be directly invoked by a user program; rather, the timer automatically starts this code on each interrupt.

```

;
; Include file for BCC180 RTOS.
;
; This file should be INCLUDED in all RTOS applications.
; It contains the macros that ease access to the RTOS itself.
;
        external      os_define,os_run,os_exit,os_cancel
        external      os_wait,os_init,os_priority
;
; This routine must be called before any other RTOS
; call is made.
;
; rt_init (no arguments)
;
rt_init macro
    call    os_init
    endm

; rt_define - Make a task known to the operating system
;
; rt_define must be called before any other commands are issued
; for the task.
;
; rt_define start,stack,bbr,priority,number
        start = start address
;
; stack = top of stack for this task
        bbr = task's bank
        priority= task's initial priority
;
        number = task number (1 to NUMTSK-1)

rt_define macro start,stack,bbr,priority,number
    ld     hl,start      ; set start address
    ld     de,stack      ; set top of stack
    ld     bc,priority*256+bbr; set priority and bank
    ld     a,number      ; task number
    call  os_define
    endm

; rt_run - Put a task in the READY state.
;
; rt_run number,rsi
        number = task number
        rsi = reschedule interval

rt_run macro number,rsi
    ld     de,rsi        ; set reschedule interval
    ld     a,number      ; set task number
    call  os_run
    endm

; rt_exit - Exit the current task
;
; rt_exit (no arguments)

rt_exit macro
    call  os_exit
    endm

; rt_priority - Set the current task's priority
;
; rtpriority priority
;
        priority= task's new priority (1 to 63)

rt_priority macro priority
    ld     b,priority
    call  os_priority
    endm

; rt_cancel - Cancel a task
;
; rt_cancel number
        number = task number (0 to NUMTSK-1)
;
rt_cancel macro number
    ld     a,number      ; set task number
    call  os_cancel
    endm

; rt_wait - Put the current task into a WAITING state
;
; rt_wait count
;
        count = number of tics to wait (1 to 32767)

rt_wait macro count
    ld     de,count      ; set count
    call  os_wait
    endm

```

Listing 2

CONTEXT\_SWITCH has two functions: decrement the RSI and wait counts of each task requiring such service, and find a task to execute.

As previously described, all registers and the stack pointer are saved for the task just suspended. That task is driven to the SUSPENDED state. Subroutine DEC\_CNTRS decrements the RSI and WAIT interval of every task in the TCB. The counts "bottom out" at zero; DEC\_CNTRS will not decrement a count below zero.

INC\_RR\_PTER increments the round robin pointer (TCBPTR). It makes sure the pointer will be aimed at a task that is available for servicing, not one that has no need for CPU time.

FIND\_TSK searches the entire TCB, starting at TCBPTR, for the highest-priority task to execute. It will select only tasks that qualify for execution time. If all tasks are the same priority, the task selected will be the one at TCBPTR, thus implementing the round robin scheduling algorithm.

READY tasks begin at the start address given in the TCB. WAITING tasks resume from the call to OS\_WAIT. SUSPENDED tasks resume from the point of suspension, with all registers restored.

Each of the RTOS service request routines was referenced in the description of task states. Listing 2 shows the macros and their calling parameters.

Well, that pretty much takes care of the task structure of the operating system. In the next issue, I'll cover the memory management aspects of multitasking, and present a sample application.

## IRS

216 Very Useful  
217 Moderately Useful  
218 Not Useful

# FIRMWARE FURNACE

## Real Numbers *Number Crunching for the 8751*

by Ed Nisley

Somehow the Intel 8051 never comes up when the conversation turns to numeric processors. An 8051 is hard to beat when you need fast execution and bit twiddling I/O, but you can't mistake it for an 80387.

Despite the fact that the 8051 ALU is only eight bits wide (it does have one 16-bit instruction!), it's still possible to handle "real" numbers. The trick is to pick a numeric format that takes advantage of the 8051's strengths and sidesteps its weaknesses.

The October through December 1988 Ciarcia's Circuit Cellar articles in BYTE Magazine described the Mandelbrot Engine Supercomputer, which is an array of Intel 8751s programmed to evaluate the Mandelbrot Set calculations. The array is controlled by an IBM AT clone that presents the results on an EGA display.

Because the Mandelbrot Set calculations require real numbers with exquisite precision, the 8051 architecture isn't one that springs immediately to mind. But we used it anyway, because an array of 8751 microcontrollers is much easier and cheaper to build than anything else. As Steve puts it, "the calculations are just a simple matter of software regardless of what kind of processor they're running on."

Where's the Point?

Although the Mandelbrot Set calculations require high precision, they do not need much dynamic

range. Those two features are often confused, but the difference is essential to making the Mandelbrot Engine work out correctly.

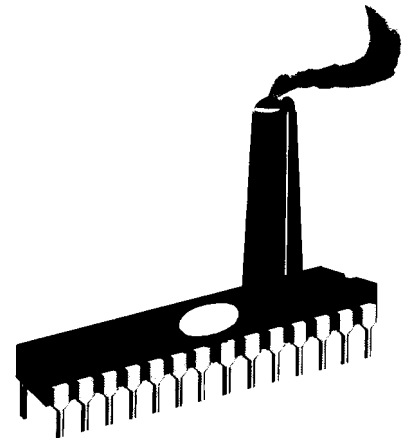
All numeric values in a computer are represented by a limited number of bits. Each bit can have two states, so an n-bit representation can have only  $2^n$  distinct values. An 8-bit number can have 256 values, a 16-bit number can have 65536 values, and so on. The key point is that once you know how many bits are used to represent a set of values, you know how many distinct numbers there can be.

Consider an 8-bit **analog-to-digital** converter: regardless of the analog input voltage range, the digital output will have only 256 distinct values. The result can be stored in a single byte without losing any precision.

If the analog voltage range is 0 to 255 volts, successive digital values are one volt apart. The value 01 hex is one volt, 02 hex is two volts, 10 hex is 16 volts, and so on to FF hex at 255 volts. The scale factor between input and output is 1 volt/count.

A more reasonable voltage range might be 0 to 25.5 volts. Now 01 hex is 0.1 volt, 02 hex is 0.2 volt, and FF hex is 25.5 volts. The scale factor now is 0.1 volt/count. Similarly, if the input voltage range is 0 to 2.55 volts the scale factor is 0.01 V/count, or 10 mV/count.

In each case there are still only 256 distinct values spread across the input range, with an obvious tradeoff between dynamic voltage



range and precision. To get a range of 255 volts you must accept 1.0-volt steps between values, and if you want 0.01-volt precision you must be content with a 2.55-volt range.

Suppose you adjusted the ADC for analog inputs between 0 and 15.9375 volts. That rather odd range gives you 256 steps at 62.5 mV/step, or 1/16 volt per count. Figure 1 shows the binary values for each input voltage. The digital value 10 hex now corresponds to 1 volt, 20 hex is 2 volts, and so on.

Imagine that there is a "binary point" after the first four bits. Any bits to the left of the point represent integers, while the bits to the right are fractions. The bit values increase by a factor of two to the left and decrease by a factor of two to the right. Once you know where the binary point is located, it's easy to read off the numeric value.

The numbers shown in Figure 1 are examples of "fixed-point" values, because the binary point is located in a fixed position in each value. A floating-point number, the kind you get when you declare a C "float" variable, has an additional group of bits to specify where the binary point occurs in the main number.

The tradeoff between fixed- and floating-point representations is simple. For a given number of bits, fixed point will have better precision because it uses all the bits to represent the value. On the other hand, floating point will have greater dynamic range because it



Full-scale range = 0 to 15.9375 volts  
 Each step = 62.5 mV

Input	Binary	Hex
15.9375	1111 1111	FF
15.8750	1111 1110	FE
...		
8.0625	1000 0001	81
8.0000	1000 0000	80
7.9375	0111 1111	7F
...		
2.0000	0010 0000	20
1.0625	0001 0001	11
1.0000	0001 0000	10
0.5000	0000 1000	08
0.1875	0000 0011	03
0.1250	0000 0010	02
0.0625	0000 0001	01
0.0000	0000 0000	00

Figure 1 -- Eight-bit values from the ADC are shown scaled to an input range of 0V to 15.9375V by using a step of 62.5 mV.

uses some bits to "shift" the location of the binary point. The representation you pick depends on what's more important: precision or range. The distinction between these systems was discussed in Ciarcia's Circuit Cellar in the November 1988 issue of BYTE, so I'll concentrate on fixed-point values in this article.

#### Bits Below Zero

What happens when you adjust the ADC to handle voltages below zero as well as above? Let's keep the same 62.5-mV scale factor as before, but twiddle the offset pot so that the input range is -8.0000 to +7.9375 volts. In real life, the digital values you get depend on which ADC chip you're using, but here we can explore some alternatives with no trouble at all.

There are several ways to represent negative numbers. Perhaps the simplest is called "signed magnitude" because there is a separate sign bit to indicate whether the magnitude is above or below zero. Figure 2 shows how the voltages would be coded in this system.

A peculiarity of signed-magnitude numbers is that there are two digital codes with a value of zero: "plus zero" (00 hex) and "minus zero" (80 hex).

zero" (80 hex). The usual convention converts a minus zero into a plus zero whenever it occurs, but it is easy to fumble a comparison and report that zero is not equal to zero.

Another problem arises when comparing values. The voltage for code 02 is greater than the voltage for code 01, but the 82 voltage is less than that for 81. A simple numeric comparison won't suffice for signed-magnitude numbers.

It would be nice if 00 hex meant 0.0000 volts and increasing digital values always represented increasing analog voltages. One way to achieve this is to keep the same hex values for 0.0000 through 7.9375 volts and use the codes for +8.0000 through 15.9375 volts for -8.0000 through -0.0625 volts. Figure 3 diagrams this approach.

If you're familiar with the 8051 (or nearly any other micro, for that matter) the sequence of digital values shown in Figure 3 should be easily recognizable. They are nothing but the integers between -128 and +127 represented in the ordinary two's complement notation used in the 8051's ALU.

In fact, the 8051's ordinary arithmetic operations will give the correct result for the corresponding analog values. This isn't magic, because we've picked the numeric representation to match up with what the 8051 does naturally. We simply don't tell it that it is manipulating "real numbers" and it doesn't know the difference.

Comparing Figures 2 and 3 will show you that the "minus zero"

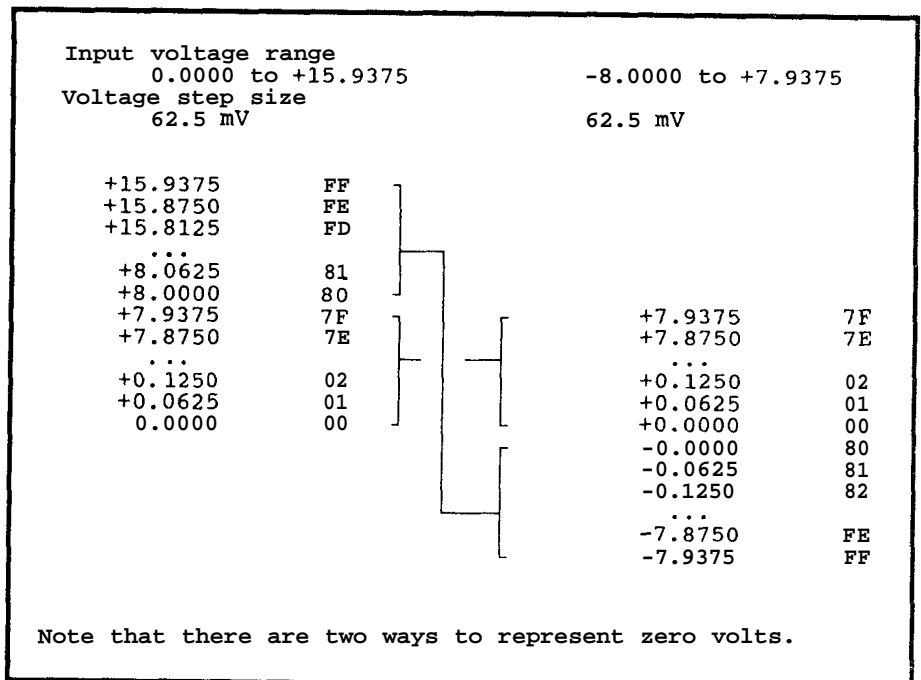


Figure 2 -- Signed-magnitude representation uses a separate bit to indicate whether a given value is above or below zero.

code produced by the signed-magnitude notation has turned into the negative code for -8.0000 volts. The only effect is that there is one negative value that can't turn into a positive value by negation.

Naturally enough, there are a variety of other numeric formats around. In fact, you will find that some ADC chips use offset binary, straight binary, and other schemes too odd to mention. Make sure you read the data sheet before you leap to dangerous conclusions!

### Home on the Range

As you should expect, though, you won't get something for nothing. Those eight bits can represent any voltage between -8.000 and +7.9375, but with a step between values of 0.0625 volts. In order to get better resolution we must reduce the overall range of numbers or increase the number of bits in each number. Conversely, to get a larger range we need either poorer resolution (bigger voltage steps between values) or more bits.

Remember that floating-point numbers don't sidestep this issue. They simply trade off resolution for dynamic range at a given number of bits. If you need a very large dynamic range and can tolerate a moderate resolution reduction, floating point may be the way to go. I'll stick to fixed-point numbers here because we need the precision.

Let's suppose that the range is OK, but we need better resolution. Replacing the ADC with one that can produce a **16-bit** result gives a step size 256 times smaller than before. The scale factor shrinks to about 244  $\mu\text{V}/\text{count}$ .

Figure 4 shows the corresponding analog and digital values. Because there are now 64K (actually 65536) numbers, the positive range goes from 0.000000 to +7.999756 volts. The negative voltages still start at -8.000000 because of the

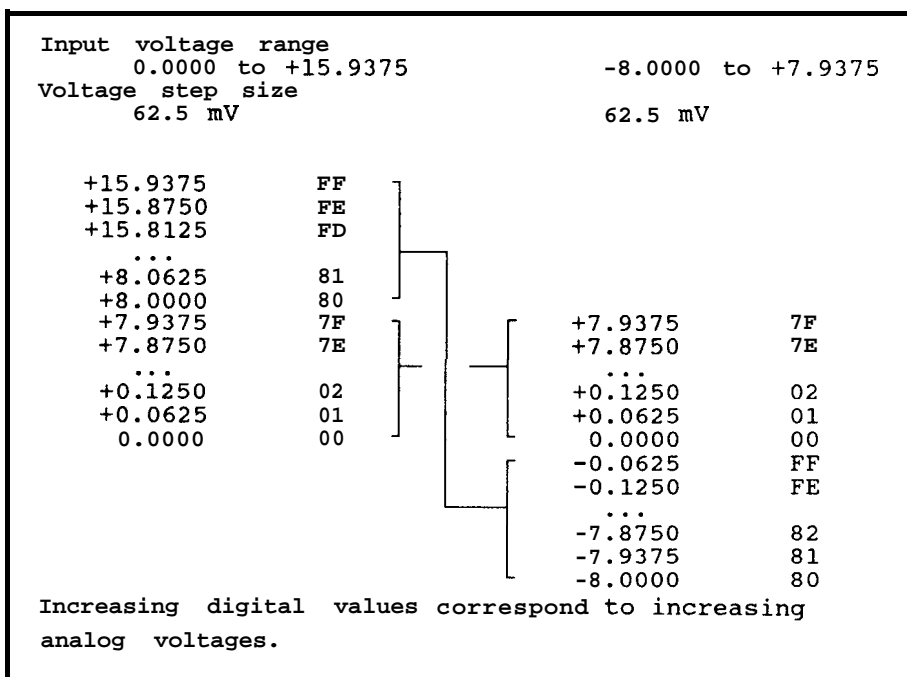


Figure 3 -- *The two's complement scheme for negative values allows 0.0 voltage to be represented as 00 hex. Note that after the representation for +7.9375 volts, the hex count "wraps around" such that -8.0000 is next in line.*

extra negative value.

Although actual analog-to-digital converters can't supply more than about 20 bits, we can continue adding more bits to our digital representation with no trouble at all. Figure 5 shows the numeric format used in the Man-

delbrot Engine. There are 64 bits, with 60 devoted to the fractional part. The overall range is -8.0 to a trifle under +8.0, with a scale factor of  $8.7 \times 10^{-19}$ . These real numbers don't have units of volts anymore because the Mandelbrot Engine works in the domain of pure

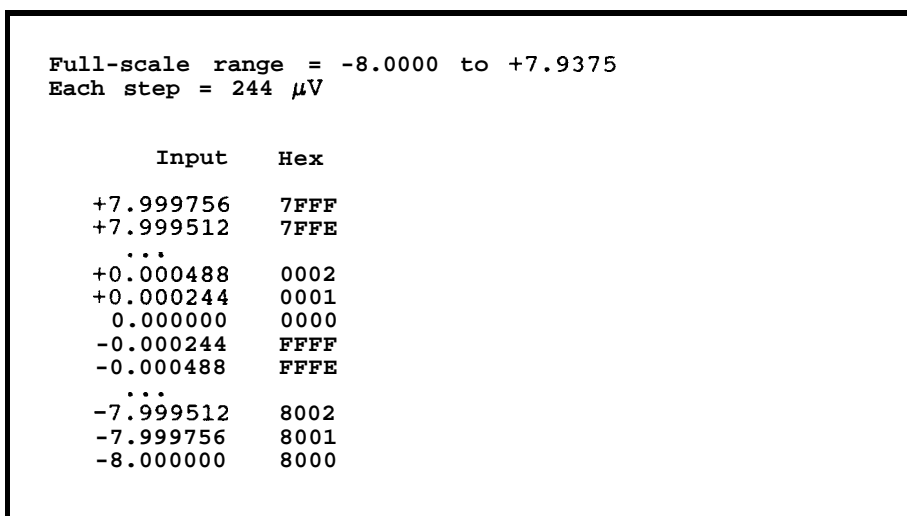


Figure 4 -- *Using 16-bit representation instead of d-bit in a two's complement scheme lets us have 256 times as many steps within the same input range.*

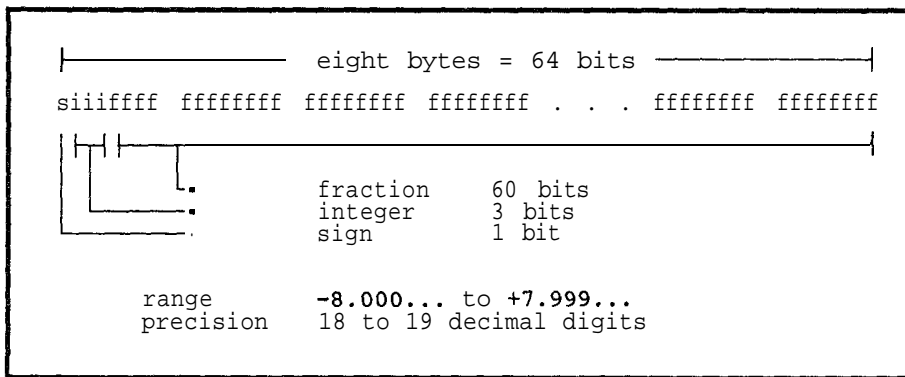


Figure 5 -- The fixed-point representation used in the Mandelbrot Engine uses 64-bit numbers, of which 60 bits are the fractional portion. Negative numbers are represented in two's complement notation.

mathematics.

But representing a number is one thing. Performing mathematical operations on these monsters is another -- remember that the 8051 ALU is only eight bits wide!

#### Extended Arithmetic

The key to extended-precision arithmetic is that the 8051 can't tell the difference between fixed-point numbers and rather long integers. The 8051's program status word (PSW) includes a carry flag that signals when the sum or difference of two bytes requires more than eight bits. The ALU can also multiply two unsigned bytes to get a 16-bit result. Those two operations are all it takes to handle the Mandelbrot Engine calculations.

The simplest operation is addition. For example, consider adding a pair of two-byte numbers: 00F8 + 010E. You'd do this by writing them one above the other

```
00F8
010E
```

and adding columns from the right. If the sum of any column exceeds F hex you would write the last digit of the sum and carry 1 to the next column. The result of all this is 0206 hex.

The 8051 ALU adds two columns (one byte) at a time, but the

process is the same. First it adds F8 + 0E to get 06 and a carry (because the result exceeds FF hex), then it adds 00 + 01 plus the carry to get 02 hex. There is no carry from that addition, so the process is done and the result fits within two bytes,

The extension to eight-byte numbers is shown in Listing 1. The two index registers RO and R1 point to the most-significant byte of the two numbers to be added and the result replaces the number pointed to by RO. The constant NUMLEN is the number of bytes in each number, which is simply 8.

All extended-precision numbers are stored with the most-significant byte in the lowest-numbered address, which is the opposite of the order the IBM PC uses. I picked this order because it makes the values much easier to read during debugging. A simple conversion routine in the IBM AT swaps the order before sending data to the array.

Because the addition proceeds from the low-order bytes upward, the first step is to adjust the pointers from the high bytes to the low ones. The 8051 can only add numbers to the accumulator, so the contents of the pointer registers must be moved into ACC, added, and then moved back.

The 8051 doesn't have the com-

plex instructions you'll find in the 80x86 family, so the loop is six instructions long. Again, the first step is to fetch the current byte from the location pointed to by RO into ACC. The ADDC instruction adds the byte pointed to by RI plus any carry from the previous addition. The result is overlaid on the source byte at RO by the second MOV instruction.

The remaining steps adjust the pointers to the next bytes (remember that higher bytes are at lower addresses) and loop for the eight counts in the B register.

You might wonder what happens for negative numbers. The answer is simple: it works just fine. It's worth stepping through a few examples on your own by hand just to make sure you understand what's going on. You can try adding some of the values in Figure 4 to see if the right answer pops out--I certainly had to do just that!

There is one slight problem, though. What happens when two numbers add up to more than +7.9999? For example, adding +4.0 and +4.0 will give +8.0, which exceeds the allowable maximum by one count. The answer for this is simple, too: the value wraps around to the maximum negative number! The addition of the high-order two bytes looks like this:

```
4000
4000
-----
8000
```

But the hex value 8000 (and the six bytes not shown here) represents -8.0, which is certainly not the right answer.

Because of the nature of the Mandelbrot Set calculations any number larger than 4.0 (either positive or negative) signals the end of the process. The routines that call *long-add* make sure that the input values won't generate an overflow, so *long-add* doesn't have to worry about error check-

```

;-----
; Add two long integers
; R0 points to the high byte of the target
; R1 points to the high byte of the source
; Mashes A and B
; Return6 R0 and R1 unchanged

long_add  PROC
PUBLIC long-add

        MOV     A,R0                ; point to end of target
        ADD     A,#NUMLEN-1
        MOV     R0,A

        MOV     A,R1                ; point to end of source, too
        ADD     A,#NUMLEN-1
        MOV     R1,A

        MOV     B,#NUMLEN           ; number of bytes to combine
        CLR     C                    ; set up for loop

L?loop   EQU     $
        MOV     A,@R0                ; pick up target
        ADDC    A,@R1                ; tack on buffer
        MOV     @R0,A                ; drop into target

        DEC     R0                    ; tick pointers
        DEC     R1

        DJNZ    B,L?loop            ; repeat for all bytes

        RET

long-add  ENDPROC

```

Listing 1 -- Extended-Precision Addition

are 32 bits in the result and we expected it to fit into 16 bits. For an integer multiplication you simply ignore the high-order two bytes, but that's not quite the answer we want here.

You first encountered this issue in elementary school when you hit (the dreaded) decimal fractions. You solved it by rote: "count up the decimal places in the multiplier and multiplicand, then shift the point over the same number in the product." Well, at least that's what I learned.

A similar rule applies to **fixed-point** numbers. Because both 16-bit numbers have 12 bits after the binary point, we count off 24 bits from the right end of the product and insert a point. The next four bits to the left are the integer part of the result. The four high-order (leftmost) bits are discarded. Figure 6(b) shows this process.

The multiplication routine can

ing.

Subtraction proceeds along similar lines and I'll leave it as an exercise for the reader. Hint: the 8051 instruction SBC (subtract with carry) may be helpful. You might also want to figure out how to take the absolute value and two's complement of a number; in a pinch, you can subtract by complementing and adding.

### Long Multiplication

Each step of the Mandelbrot Set calculation requires four **eight-byte** multiplications. At first glance you might think that the process is just as simple as addition because the ALU can multiply two **8-bit** bytes. That's not how it works out ...

Let's start with the same **16-bit** numbers: **00F8 x 010E**. Figure 6(a) shows the intermediate steps required to do this by hand. The product is **0001 0590** and the first problem should be obvious: there

6a -- 16-bit multiplication using byte multiplies

```

                                00 F8
                                01 0E
                                -----
F8 x OE =                       0D 90
00 x OE =                       00 00
F8 x 01 =                       00 F8
00 x 01 =                       00 00
                                -----
                                00 01 05 90

```

6b -- Aligning the binary point

```

                                0.0 F8  <- 12 bits after point
                                0.1 0E  <- 12 bits after point
                                -----
                                0D 90
                                00 00
                                00 F8
                                00 00
                                -----
                                00.01 05 90  <- 24 bits after point
                                [-----]
                                <- 16 bits of result
                                0.0 10    <- the final result

```

Figure 6 -- The first figure shows a problem inherent in multiplying 16-bit fixed-point numbers: the result is a 32-bit number. The second figure shows the "fix" for this problem that was used in the Mandelbrot Engine.

extract some information from the "excess" bits. For example, the 16-bit product can be rounded based on the contents of the low-order 12 bits. The high-order four bits can indicate whether the product is too big, and, if so, the code can substitute the maximum possible 16-bit number.

There is yet another complication that isn't obvious from this simple example. Each and every byte multiplication generates a 16-bit product that must be added into the partial product at the correct spot, but those additions can cause a carry that must be propagated all the way to the most-significant bit. Embedded in each multiplication is a carry loop as well.

Finally, unlike addition and subtraction, the multiplication instruction works only for positive (and zero!) values. The Mandelbrot Engine code determines the sign of the product from the signs of the incoming values, then takes their absolute values. The multiplication takes place in a temporary 128-bit buffer and the result is rounded, shifted, clamped, and re-signed in place.

The source code for *long\_mult* is so, well, long that Steve and Curt turned pale when I suggested printing it. We compromised: it's available on the Circuit Cellar BBS in the DRIVER.ARC package along with the source code for the IBM AT control program. The remainder of the 8051 code isn't available because of licensing agreements, but you're sure to see interesting snippets of it in upcoming columns.

The 8051 has so few useful registers that adding the overhead for all the multiplication and carry propagation loops didn't make any sense. The alternative is simple: write the code as a straight-line routine without loops. But straight-line source is awkwardly bulky, so I used the preprocessor facilities of AVMAC to build "compile time" loops to generate

the code.

Listing 2 shows the core of the multiplication code. It takes effect after the partial product for the low-order multiplicand byte is

finished. There are two nested loops, one to handle the remaining multiplier bytes and the other to handle all the multiplicand bytes every pass. The %IF statements

```

The 16-byte product buffer is defined by these macros:
;-----
; 16-byte buffer for extended-precision multiplies
; Higher-order bytes are at lower addresses
;--- 40H
mpy&N      %FOR      N = NUMLEN-1 TO 0 BY -1
           DS         1
           %ENDFOR
prod&N     %FOR      N = NUMLEN-1 TO 0 BY -1
           DS         1
           %ENDFOR

Only the core of the multiply routine is shown below.
RO points to the multiplicand.
The multiplier is located in the buffer at mpy0 through mpy7
and is replaced by the partial products.
;-----
; Rub remaining multiplier bytes across multiplicand
           %FOR      J = 1 TO NUMLEN-1 ; multiplier index
           MOV       DPL,mpy&J        ; set up multiplier byte
           MOV       mpy&J,#0         ; and clear for results
           JB        compute,m&J&top  ; check for cancel flag
           JMP       restart          ; ... exit if off
m&J&top    EQU       $
           %FOR      I = 0 TO NUMLEN-1 ; multiplicand index
m&J&&I     EQU       $
           MOV       A,@RO            ; next multiplicand byte
           DEC       RO
           MOV       B,DPL            ; multiplier byte
           MUL       AB
           ADD       A,prod&J-&I      ; combine low byte
           MOV       prod&J-&I,A
           MOV       A,B
           ADDC      A,prod&J-&I-1    ; combine high byte
           MOV       prod&J-&I-1,A
           %IF      &I LT NUMLEN-1 .. no carry past multiplier byte
           %FOR      N = 2 TO NUMLEN-&I : ... but all remaining bytes
           JNC      noc&J&&I
           CLR      A
           ADDC     A,prod&J-&I-&N
           MOV      prod&J-&I-&N,A
           %ENDFOR
           %ENDIF
noc&J&&I   EQU      $
           %ENDFOR ; multiplicand loop
           %IF      &J NE NUMLEN-1 ; special reset for lest byte
           MOV      A,RO ; point to last byte
           ADD      A,#NUMLEN
           MOV      RO,A
           %ELSE
           INC      RO ; point to first byte
           %ENDIF
           %ENDFOR ; multiplier loop,
haveprod   EQU      $

```

Listing 2 -- Extended-Precision Multiplication

control the length of the carry propagation chain.

The bottom line of all this is that an eight-byte multiplication is a very expensive process. After a good deal of tinkering and fiddling with the code, the Mandelbrot Engine can compute one iteration of the formula in about 5 milliseconds.

While 5 milliseconds per iteration sounds painfully slow, remember that the point of the project was to demonstrate that a large number of these simple processors can be faster than any single processor, no matter how fast. For example, 64 processors can drop the average time to 78  $\mu$ s per iteration; and 256 processors can push it under 20 ps. Communication and overhead will prevent the array from reaching those ideal values, but the principle is still

valid: there is strength in numbers!

### Fixing Your Points


Although the ready availability of math **coprocessors** makes floating point the natural choice for many PC projects, the case isn't closed. You may find that floating point isn't the ideal solution for your code, particularly if you don't have enough bits ... and nobody ever does. For projects needing lots of precision over a small dynamic range, try some fixed-point math.

### Future Directions

This column marks Firmware Furnace's first anniversary! In the past year we've explored some 8051 code, checked out the IBM PC timer, fiddled with buttons and

joysticks, and gotten better acquainted with the DDT-51 development system.

Here's the **64K-byte** question: what would you like to see for the next year or so? I've heard cogent arguments for less PC coverage, more PC coverage, less 8051 code, a full-blown DDT-51 project, high-level-language projects (huh?), and so forth and so on.

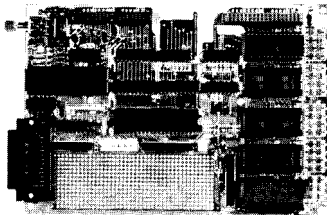
If you've got strong feelings one way or another drop a letter, post a Circuit Cellar BBS message, or flash an **EasyPlex** on CompuServe (74065,1363). There are about 25,000 of you reading INK nowadays, so I should get a flood of suggestions! 

### IRS

**219** Very Useful  
**220** Moderately Useful  
**221** Not Useful

# SIBEC-II

*The ideal solution for embedded control applications and stand-alone development.*



- Intel 8052AH BASIC CPU
- Serial printer output and 5, 8 bit I/O ports
- 5 in.<sup>2</sup> prototyping area
- Memory: 8K RAM, 16K EPROM, expandable to 48K
- Program in BASIC, assembly or a combination of both
- PROM programmer; ZIF socket for 2764 or 27128 EPROM
- Interrupt handling capability
- Built to exacting standards and warranted
- \$228.00 including documentation (quantity 1)

Inquire about our **PK51**. The 80518052 product development kit for the IBM-PC/XT/AT. Includes the SIBEC-II, power supplies, cross-assembler, and much more. \$595.

**Call now! 603-469-3232**



**Binary Technology, Inc.**

Main Street • P.O. Box 67 • Meriden, NH 03770



## WRITE FOR INK!

Writing technical articles may **not** make you rich and famous but it might be just the incentive to finish that 100-MIPS computer you started last summer. Or, if your expertise is software, perhaps it's time you presented your talents to the world.

Unlike most narrowly **specialized** publications, Circuit Cellar INK's charter is to cover a wide variety of hardware and software technology and ideas.

Send your project outline to:

**Curtis Franklin, Jr.**  
Circuit Cellar INK  
**P.O. Box 772**  
Vernon, CT 06066

or contact him on the Circuit Cellar BBX at (203) 871-1988.

### Circuit Cellar Books

Circuit Cellar INK author often refer to previous Circuit Cellar **articles**. These past articles are available in book form from Circuit Cellar Inc., 4 Park St, Suite 12, Vernon, CT 06066. Ciarcia's circuit Cellar Vol. I covers **articles** in BYTE from September 1977 through November 1976. Vol. II covers December 1976 through June 1980. Vol. III covers July 1980 through December 1981. Vol. IV covers January 1982 through June 1983. Vol. V covers July 1983 through December 1984. Vol. VI covers January 1985 through June 1986.

# CONNECTIME *Excerpts from the Circuit Cellar BBS*

THE CIRCUIT CELLAR BBS  
300/1200/2400 bps  
24 hours/7 days a week  
(203) 871-1988 -- 4 incoming lines  
Vernon, Connecticut

*The message base of the Circuit Cellar BBS is now available on disk. See page 62 for details.*

Well, we've had some exciting times around here recently. Version 2.1 of TBBS has arrived, and with it a host of changes to make the sysop's (read "my") life easier and to make your stay on-line an easier and more efficient one.

Most obvious to those who write a lot of messages is the ability to use either ASCII or a binary error-checking protocol to upload message text which was prepared off-line. Now you can spend time off-line without the clock ticking to prepare a thought-out message, then upload it without the hassle of tricky delay insertions or checking for noise-induced errors. Leading spaces on lines are left in place. Formerly, any leading spaces were removed, making a once-nicely formatted message come out as garbage. Finally, message "threading" has been improved to make groups of related messages easier to follow.

Those who do a lot of file transfers will also notice some improvements. More protocols have been made available. In addition to the old standby ASCII, XMODEM, and YMODEM (XMODEM/IK), we now have YMODEM Batch (True YMODEM), SEAlink, KERMIT, and SuperKERMIT (sliding windows). For the novice, more extensive help is available which describes each of the available protocols in detail. If you see an interesting-looking ARC file, the system will show you a list of files it contains. If you're only interested in files uploaded since your last call, you can request a list of "new" files. TBBS 2.1 also introduces a new concept in file organization which I'll be implementing in the weeks to come.

There are many more changes and additions in version 2.1, some major, like those I've listed, and some minor, but far too many to list here. I'll be slowly making changes to the CCBBS to implement some of the new features and modifying the way existing commands work. There is even a provision for full-color screen-oriented menus with graphics characters for systems that support such features. Stay tuned. The best is yet to come.

One method for doing home (or factory) control is distributed intelligence. Processors located around the

premises take care of local details, communicating general status and other requests to a central host via some form of network. The following is a discussion of one person's attempt at a control network.

Msg#: 8452 from JEFF JENSEN

Mark, in your home control work, have you developed any home LAN communication protocols? I have been rolling the requirements of a two-way LAN around and wondered if you have implemented anything special in your systems that might be important. I have several design considerations that may or may not be important in this environment, and I'm finding that a fuller protocol with routing, addressing, error detection, control, and data packeting gets to be quite a lot of overhead for slower networks.

Msg#: 8484 from MARK LAMPKIN

Jeff, the way the system started was as a simple protocol. As the system was implemented, the protocol developed a thyroid condition and started growing out of proportion to the actual needs of the system. Now it is at a simple but powerful (and useful) level. The actual system is more of a token-pass ring network. To keep fibering (my term for wiring, but with fiber optics) to a minimum, the network is a closed, unidirectional ring. Each controller is listening to its upstream neighbor. If its address matches the second byte in the packet, it responds to the third byte (the control word) and creates a packet of its own. The basic command structure is:

```
Byte 1    -- STX
Byte 2    -- Address
Byte 3    -- Command
Byte 4    -- Packet Length
Byte 5    -- Data
Byte 5+N  -- Data
Byte 5+N+1 -- ETX
```

So far the communication error rate has been zero so there is no checksum. If the error rate starts to creep up, simply add some checking. The whole network just keeps passing the message until it is processed and a response has been taken. Controller #1 always starts the token.

Msg#: 8604 from RON WILSON

Your protocol looks a lot like HDLC. Motorola (and others) make a chip that does exactly what you described.

Msg#: 8690 from MARK LAMPKIN

Ron, it is quite similar, eh? However, those chips are harder to come by, and not as cheap as me and a little software.

**Msg#:** 8517 from JEFF JENSEN

Your protocol looks much like what I have come up with, except I had also included originator address and a preamble. I've spent too much time looking at LAN protocols. Does your net use a single micro type or have you got a bunch of micro families represented? I also wondered what data rate you are running at, which would affect the error rate and impact of message or packet **size**.

One of my interests would be to allow all three types of home functions on the same wire -- monitoring (security and status), control, and peer-to-peer communications. It seems that to keep the coat of the system down, a transaction-oriented approach (**small, self-contained** message packets) would be better than massive message and bulk transfer.

**Msg#:** 8593 from MARK LAMPKIN

Jeff, so far on my network I have three **80C85s**, two **1802s**, six **68705P3s**, two **68705R3s**, one 280, four **6809s**, and two **680x0s**. Kind of the **Heinz 57** approach. In a home system, security is not of the greatest concern (to me), and taking into account the security system I have implemented, the loop turn-around time is from S-400 milliseconds on the average. I'm presently running 38.4 kbps, but have considered going up to **76.8k**. The reason for this is a project I'm considering. I'll **need a little more dedicated task time** in a new node; not so much time can be spent loop processing.

**Msg#:** 8469 from KEN HOWELL

This is something that always fouls me up in home **LANs** -- in your **scheme**, how does a controller other than **#1 "grab" a slot to send** its own signal without interrogation from controller **#1**?

**Msg#:** 8689 from MARK LAMPKIN

Ken, the interesting part of this type of system is, as with anything, if it is implemented correctly, it's crash-proof. The way to make it crash-proof is to examine the network **topology**. It's a ring, and each controller receives the message. If the address doesn't match, the node relays the message to its downstream neighbor. When a controller wants to send a broadcast to another controller, it waits for an incoming message, buffers the received message, inserts its own outgoing message, and finally completes its task by relaying the buffered message it last received. Quite simple yet eloquent. I wish I could take claim for the concept.

For network start-up, controller **#1** comes on-line and delays the calculated maximum loop delay. If no message has been received within that time frame, it sends a token message to itself containing dummy data under its own address. This checks loop integrity on the first pass, and after the first pass the same message is continued to be sent to enable the other stations to achieve a time slot to get on the network.

So are the basics of my LAN. Any diagnostic errors are displayed on the host. It just so happens that, in my command structure, a command byte of "OOH" is a broadcast of the data contained in the packet. This packet then is considered as a network display, meaning that if a node has display capability (i.e., something readable or decipherable by the imperfect human sensory capabilities), the packet data is displayed. This is then a broadcast to all the operating nodes to pinpoint the failed module. Simple, eh?

**Msg#:** 8600 from KEN HOWELL

Well, that certainly explains things! The Motorola **6870x** series implements a built-in protocol, where the bus can actually "sleep" until activated by messages. I don't think it is as robust as the protocol you describe.

**Msg#:** 8614 from MARK LAMPKIN

Ken, the Motorola protocol of which you speak is in the 6801 family of products (i.e., 6801, 6803, 68701, etc.). I'm using the 6805 family of products -- no built-in UART. I'm doing the UART and protocol all in software. The 6805 family of processors is my favorite to work with. Straightforward memory map, true bit manipulation, and anything from **321/O pins to on-board A/D** or phase-locked loop. Really a neat chip.

**Msg#:** 8622 from KEN HOWELL

I picked up a few of **them when** Jameco was having a **sale** on them. I've also got a number of the Motorola application notes on the family. I'm in the process of building a programmer for my Amiga for these little guys, and look forward to when I can make one sing. I am uploading today the 6870x assembler. I assume that you have something already.

By the way, **regarding your home LAN**, what happens if one node goes down? Does this break the ring, and thereby ruin the integrity of the LAN?

**Msg#:** 8630 from MARK LAMPKIN

Ken, presently the LAN goes down from the culprit node on. I am moving into a new home in about two weeks and I get to start from scratch. My present LAN is all fiber optic, so internal to each system is a watchdog timer which in most cases takes care of the bad-node problem. When and if one goes terminal (pun? -- not much of one), all the operational nodes display the fault. Then comes human intervention. In 47 months of operation, I've had only two terminal terminals -- an acceptable number for me.

**Msg#:** 8649 from KEN HOWELL

Well, that's not a bad history. I've seen **LANs** that are in a ring configuration, but the nodes are only "listening" to the ring and do not represent a break in the ring. This approach won't crash the way your LAN would, but I think the interfacing details become stickier.

**Msg#:** 8676 from MARK LAMPKIN

Ken, interfacing to a LAN with the type of architecture you are describing is best done as a token-pass-type mastery. This is the way some well-known (in the auto industry) highways work. The problems encountered are software overhead, crashing, and, to a large extent, noise. The most efficient **comm** systems will have very strict rules and efficient message packets -- a small sacrifice for a reliable system. Generation **#2** of my system will become fail-safe.

**Msg#:** 8687 from KEN HOWELL

How can you have a fail-safe system where the integrity of the LAN depends on perfect operation of each node?

**Msg#:** 8734 from MARK LAMPKIN

Ken, my new system is going to be transformer-coupled to the network. The same mode of operation will still be in place, but an addition of a node timeout will signal to the downstream tap that the upstream is dead.

**Msg#:** 8668 from JIM NELSON

Are you familiar with **CEBus**, the Consumer Electronics Bus?



It's an EIA standard for communications among consumer electronics products and home appliances being worked out even as we sleep.

**Msg#: 8673 from MARK LAMPKIN**

Jim, I have been with companies that used many different standards and tried to develop standards. I was on the Honeywell MAP committee and others. The whole development of this system was an idea of my own to develop a **working system** without the many, many **layers** of sophistication that a group consensus operation will develop. KISS (Keep It Simple, Stupid) is a much more powerful tool than all the error-checking code and redundancy can buy.

**Msg#: 8731 from JEFF JENSEN**

Do any of the nodes perform diagnostics or have a hardware watchdog **timer** to reset them? One approach to a self-diagnosing ring would be for the timeouts on each node to cause a packet to forward to a designated node and have it log everyone that responds. The terminal terminal would be the **first** missing node, or the **first** node in the loop to log a message.

One question I meant to ask earlier: does one node act as ring master and issue a token? In that case, if the token dies, then does the ring master time out **first** and send a new token?

**Msg#: 8757 from MARK LAMPKIN**

Jeff, the only master on my present ring is but a temporary thing. It is needed to start the **first** message on the ring. After that it is the domino effect, unless the master never initiates the **first** command. This is to be solved on my new network in three to four weeks.

**The bane of any complex, well-stocked home entertainment center is how to cross-connect all the equipment. Should the output of VCR 1 go to VCR 2, the local television, or the video distribution system? The following thread concerns some of the design issues related to a switch box designed to solve such problems.**

**Msg#: 8514 from VINNY RUSSELL0**

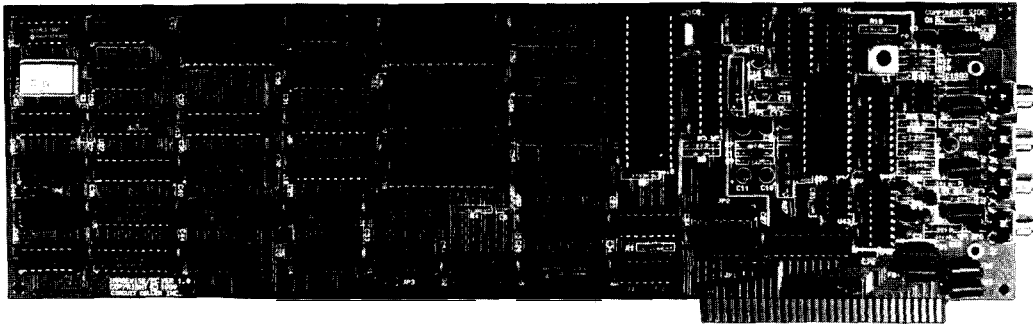
Steve, I have run into a problem that you may be able to solve, and while we're at it start a whole new project. The problem is I want a home computer system with information distributed to any TV set in the house.

I see no problem with taking an IBM PC CGA output and modulating it to RF. The PC will have software running any kind of data I desire. Right now it is connected to my weather station and X-10 control device (from Heathkit). In my house I have six RF video feeds. I would like to switch from the following RF sources: 1) raw unscrambled cable (this is all the channels but HBO), 2) HBO (unscrambled using cable box), 3) VCR RF output (channel 3), 4) computer RF output. The goal of this system is to walk up to any TV in the house and select one of the inputs.

One system I've found is an RF/video distribution system from a company called Channel Plus and consists of "Universal Video Channel Plus Multiplexers." The system allows you to assign specific UHF channels to RF/video sources. However, the cost for the units is: 1 RF channel, \$189.00; 2 RF channels, \$309.00; and 3 RF channels, \$489.00.

I read your article in the February 1986 BYTE about the Audio/Video Multiplexer. However, this will not work with the RF outputs from cable TV, cable TV box, VCR, laser disc RF outputs, and

## Micromint now makes affordable Video Digitizing even better with ImageWise/PC™



The company that made video digitizing affordable now makes affordable digitizing even better with ImageWise/PC™. Bring your reports, graphics, security system, or video application up to the new standard in cost-effective gray scale video digitizing with the new ImageWise/PC™.

- Digitize any NTSC, PAL, or SECAM video source!
- Up to 256x255 resolution with 256 level gray scale!
- True frame grabber - digitizes in 1/60 second!
- Digitizes 30 frames per second!
- Composite video output!
- Can display digitized pictures on EGA or VGA!
- Advanced overlay and split screen capabilities!
- Digitized images compatible with paint and desktop publishing programs!
- Modify, enhance, display, and print images using sophisticated ZIP Software Included with every ImageWise/PC!

**ImageWise/PC™**  
an affordable  
**\$795.00**

Order by: TEL: (203) 871-6170  
FAX: (203) 872-2204  
TELEX: 64333 1



MICROMINT, INC. - 4 Park St., Vernon, CT 06066

computer RF. I'm wondering if you could design a **computer-controlled RF switcher** that will handle perhaps an **8-input/8-output** system?

**Msg#: 9451** from STEVE CIARCIA

I don't plan on building an RF switch because I already have one. An **8-by-8** RF/audio/video (actually, I don't think it switches eight RF channels) mux called the Component Coordinator. It sells for about \$795 and used to be frequently advertised in all the video and stereo mags. That's the best I can suggest unless you want to spend a lot of money on coax relays.

**Msg#: 9484** from JIM NELSON

My name's emblaeoned on the **PCBs** in your Component Coordinator, Steve. I was the chief electronic engineer on that project at Video Interface Products. There were only two electronic design engineers including me, but that's just an indication of how hard we worked. It's a mix of high and low tech, especially in the mechanical department, where the technology ranges from photochemical machining to **crazy** glue.

It switches three input RF channels (8, 9, 10) by six output RF channels (1-6) and seven input video and stereo audio channels by eight output channels. RF input channels 9 and 10 use Omron RF relays (flat to about **700 MHz**); diode switches are used on the other RF input column and to switch the RF-modulated baseband audio/video source into RF outputs 1 and 6.

By the way, as a Component Coordinator buyer, you're in the company of Ford Aerospace (who bought forty) and Burt Reynolds (who bought one). We sold several thousand of those, but Video Interface folded last year.

Suddenly I don't feel like such an unknown quantity around here.

**Msg#: 9510** from STEVE CIARCIA

Well, I'll be. My Component Coordinator (I agree that it is a **mix** of sophistication and kludge) fits neatly into my Nakamichi A/V system. While I designed my own A/V **mux**, the CC was packaged more appropriately for my needs at the time. I don't use any of the RF switches and only switch audio and direct video.

I remember talking to the people at Video Interface Products and they weren't very nice. They approached sales as if they were doing me a favor selling it to me. Good thing it worked. I sure wouldn't have wanted to deal with those turkeys for service. What were the details of their demise? Finally, since you were the designer, perhaps you might have a schematic that I could have (or one that I can copy and return to you) just in case this thing ever bites the big one. With all the lightning problems I've had, I've been lucky so far.

P.S., How do you sell several thousand and go out of business?

**Msg#: 9551** from JIM NELSON

It's interesting, and typical, that you don't use the RF section at all. The first version of the CC (named the **FromTo**) was a **10 x 8 RF-only** switch. Although it won a design engineering award at the 1983 June CES (for a photo, see *Radio & Electronics*, Sept. 1983, p 50), the RF-only switcher neverreached production; the package and the name were changed.

The RF matrix in the model you have was the most expensive subset of the CC's production cost. If you've opened it up, you probably noticed that the RF outputs are connected to six discrete **PCBs**. Each of those output channel **PCBs** contains a **photochemically** machined RF shield and a pair of equally pricey Omron **G4Y** RF (**104-dB** isolation @ **250 MHz**) relays.

Most people bought it for the audio/video matrix. It was a price/performance steal. The unit was designed to fit the whim of Video Interface's owner; no amount of reasoning could convince him to introduce an A/V only machine or a simpler machine because it violated his "inner image" of the market. Only toward the end of **V.I.P.'s** corporate life did they begin creatively exploiting its **poten-**

tial by doing things like selling it with BNC jacks -- and cranking up the price.

Less than ten units sold out of the entire production of about 2000 units were returned for repair. That's why you've never had problems. In fact, I guess I am why you've never had problems with that unit. <big grin>

**Msg#: 9560** from STEVE CIARCIA

Thanks, Jim. I'd love to hear more. BTW, I did open it and it did seem to have a lot of trash in it. My only complaint is that the **matrix LED** display is much too dim, but I didn't want to try goosing it because it looked like a pretty small transformer (don't need any fires in the entertainment room).

**Msg#: 9590** from JIM NELSON

That signal transformer ran cool as a refrigerated wombat, Steve. It had lots of headroom. In the Component Coordinator, CMOS chips outnumber the others on the boards 28 to 27, so the system as a whole runs pretty cool, too. Of course, it may just run cool because we were able to force most of the energy to be dissipated as **EMI**. :-)

The **LEDs** are arranged in a time-division-multiplexed **8 x 16** matrix. Each LED is pulsed at **40 mA** with a duty cycle variable from about 4% through 12%. I used UDN2983 Darlington packs driven by a **1/8** decoder to source current to the eight scanned columns of **16 LEDs**. We matched LED brightness by using DS8859 latched programmable constant-current sinks tied to the cathodes of the **16-bit** addressed **LEDs** in each column. There is nothing that can be easily done to increase the brightness of the display; there are no resistors to change.

Regards, Jim

---

**The Circuit Cellar BBS runs on a IO-MHz Micromint OEM-286 IBM PC/AT-compatible computer using the multiline version of The Bread Board System (TBBS 2.1M) and currently has four modems connected. We invite you to call and exchange ideas with other Circuit Cellar readers. It is available 24 hours a day and can be reached at (203) 871-1988. Set your modem for 8 data bits, 1 stop bit, and either 300, 1200, or 2400 bps.**

IRS

222 Very Useful

223 Moderately Useful

224 Not Useful

#### SOFTWARE and BBS AVAILABLE on DISK

##### Software on Disk

Software for the **articles** in this issue of Circuit Cellar INK may be downloaded free of charge from the Circuit Cellar BBS. For those unable to download files, they are also available on one **360K, 5.25" IBM-PC-format** disk for **only \$12**.

##### Circuit Cellar BBS on Disk

Every month, hundreds of information-filled messages are posted on the circuit Cellar BBS **by people from all walks of life**. For those who can't log on as often as they'd like, the text of the **public message areas** is available on disk in two-month installments. Each installment comes on three **360K, 5.25" IBM-PC-format** disks and costs just \$15. The installment for this issue of INK (**January/February 1989**) includes all **public** messages posted during November and December, **1988**.

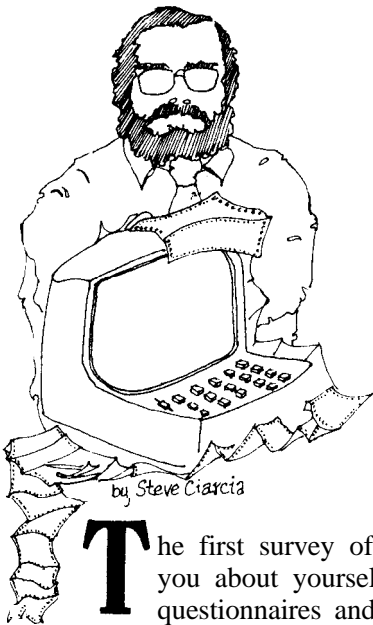
To order either Software on Disk or Circuit Cellar BBS on Disk, send check or money order to:

**Circuit Cellar INK  
Software (or BBS) on Disk  
P.O. Box 772  
Vernon, CT 06066**

or use your **MasterCard** or Visa and call **(203) 8752199**. Be sure to specify the issue number of each disk you order.

# STEVE'S OWN INK

## First INK Reader Survey



**T**he first survey of Circuit Cellar INK subscribers is finished, and I thought I'd use this column to tell you about yourselves and about what you told us to put into the magazine. We sent out 2000 4-page questionnaires and so far we've gotten over 500 of them back. That's a tremendous return for a survey, and it tells us that you have an interest in Circuit Cellar INK that goes way beyond the average reader/magazine relationship.

The short description of a Circuit Cellar INK reader runs like this: You're a successful, experienced professional who enjoys the satisfaction of problem solving both at work and at home. Now, let me flesh the description out a little bit. First, I'm going to talk about men, since about 99% of you are male. Next, just about half of you say that you are involved in engineering, even though about half of you engineers have job titles that place you squarely in the management camp. Over half of you say that you've been involved with computers for over 10 years, and 93% have been in the game for more than 5 years. Whether your length of experience includes college I don't know, but I do know that over two-thirds of you have at least a bachelor's degree. All that experience and education seem to be paying off, too, since over two-thirds of Circuit Cellar INK readers have incomes that make them eligible for the Yuppie Hall of Fame.

Now on to the important stuff. It didn't surprise us to find that 97% of Circuit Cellar INK readers own a computer. It was a little surprising to see that only 1 out of every 5 of you stopped at a single computer. Nearly two-thirds of you have an IBM XT or AT, while close to half of you say that you own a single-board computer. Other computers mentioned run the gamut from old Ohio Scientific 6502 machines through S-100 boat anchors to the latest Macintosh 11s and **80386-based** computers.

While I see Circuit Cellar INK readers as hardware designers par excellence, 90% of you admit to writing software as well. It's nice to see that you aren't afraid to tackle both sides of a project. When you get down to programming, assembler, BASIC, FORTRAN, and C (in order of popularity) are the tools you choose. Speaking of projects, I was impressed to see that 77% of Circuit Cellar INK readers design and build applications for personal use. In addition, over two-thirds of you say that you're planning to build an electronics kit in the next twelve months. With all this designing and building, it looks like there'll be a whole lotta solderin' going on.

In addition to telling us about yourselves, the survey let you tell us what you think about Circuit Cellar INK. The most important thing we saw (and the most gratifying) is that you think we're doing a good job. In comment after comment, you told us that you want a very technical, practical, solution-oriented magazine with an emphasis on applications. Circuit Cellar INK readers like the humorous introductions to some of the articles (OK, my articles) in the magazine, but want to make sure that the technical content stays solid. You most certainly don't want "Circuit Cellar INK Looks at 135 AT Clones" articles (don't worry) and you would like to see tutorials covering practical aspects of computer application design and construction. You want more embedded control and microcontroller applications, and could do without glowing descriptions of technology that no one can afford.

The survey helped us get a better picture of who you are, and gave us some clear directions on making Circuit Cellar INK the magazine for serious designers and builders. All in all, the magazine you say you want is exactly the one that we're planning to continue. To everyone who responded to the survey, thanks.

Steve Ciarcia