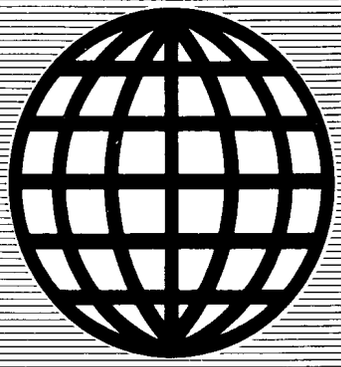


The COMPUTER JOURNAL

Programming - User Support
Applications

Issue Number 51 July / August 1991 US$3.95

Introducing the YASBEC
Floppy Disk Alignment with the RTXEB and Forth
High Speed Modems on Eight-Bit Systems
A Z8 Talker and Host
Local Area Networks
UNIX Connectivity On The Cheap
The PC Hard Disk Partition Table
A Short Introduction to Forth
Real Computing
Z-System Corner
PMATE/ZMATE Macros
Z-Best Software
Stepped Inference as a Technique for Intelligent Real-Time Embedded Control
The Computer Corner

ISSN # 0748-9331

Now $4.95 Stops The Clock On Over 100 GEnie Services

For the first time ever, enjoy unlimited non-prime time* usage of many popular GEnie℠ Service features. For just $4.95 a month. Choose from over 100 valuable services including everything from electronic mail and stock closings to exciting games and bulletin boards. Nobody else gives you so much for so little.

You can also enjoy access to a wide variety of features like software libraries, computer bulletin boards, multi-player games, Newsbytes, and the Computer Assisted Learning Center (CALC) for just $6.00 per non-prime hour for all baud rates including 2400. That's less than half of what some other services charge. Plus with GEnie there's no sign-up fee.

Now GEnie not only gives you the information and fun you're looking for. But the time to enjoy them, too.

Follow these simple steps.

1. Set your modem for half duplex (local echo), at 300, 1200 or 2400 baud.
2. Dial toll free 1-800-638-8369. Upon connection, enter HHH.
3. At the U#=prompt, enter XTX99486,GENIE then press RETURN
4. Have a major credit card or your checking account number ready.

For more information in the U.S. or Canada, call us voice at 1-800-638-9636.

TCJ readers are invited to join us in the CP/M SIG on page 685 and the Forth Interest Group SIG on page 710. Meet the authors and editors of *The Computer Journal!* Enter "M 710" to join the FIG group and "M 685" to join the CP/M and Z-System group.

We'll meet you there!

JUST $4.95

Moneyback Guarantee

Sign up now. If you're not satisfied after using GEnie for one month we'll refund your $4.95.

*Applies only in U.S. Mon.-Fri., 6PM-8AM local time and all day Sat., Sun., and select holidays. Prime time hourly rates $18 up to 2400 baud. Some features subject to surcharge and may not be available outside U.S. Prices and products listed as of Oct.1, 1990 subject to change. Telecommunications surcharges may apply. Guarantee limited to one per customer and applies only to first month of use. GE Information Services, GEnie, 401 N. Washington Street, Rockville, MD 20850. © 1991 General Electric Company.

The Computer Journal

Founder
Art Carlson

Editor/Publisher
Chris McEwen

Technical Consultant
William P. Woodall

Contributing Editors
Bill Kibler
Matt Mercaldo
Tim McDonough
Frank Sergeant
Clem Pepper
Richard Rodman
Jay Sage

The Computer Journal is published six times a year by Socrates Press, P.O. Box 12, S. Plainfield, NJ 07080. (908) 755-6186

Subscription rates• Within US: $18 one year (6 issues), $32 two years (12 issues). Foreign (surface rate): $24 one year, $44 two years. Foreign (airmail): $38 one year, $72 two years. All funds must be in U.S. dollars drawn on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: *The Computer Journal*, P.O. Box 12, S. Plainfield, NJ 07080, telephone (908) 755–6186.

Registered Trademarks

TCJ The Computer Journal

Issue Number 51 **July / August 1991**

Editor's Desk

By Chris McEwen

I had a bit of a surprise when I called the printers to tell them I was on the way in with this issue. "We close shop this time every year for vacation. Can get you in on July 8, though. Didn't anyone tell you?" Well, no. No one told me, and with a printing schedule of 10 days, I don't need to tell you this issue is late getting out of the starting blocks. You know that already! Getting an issue out on a preset date is frustrating me! Meanwhile, I have taken the opportunity to add some last minute comments to this column, which you will find in italics.

We have gained a raft of new readers in the last few months. I thought this might be the appropriate time to talk a little about our industry and where we all fit in.

Pocket Ventures

Let us start with a basic assumption. You are not Motorola, and I am not Zilog. Collectively, we would not equal one percent of Intel's net worth. Facts are facts, friends, we are small fish. *TCJ*, being a small fish, caters to such. This is not bad. In fact, it is very, very good. Major corporations take on a personality of their own, often at the expense of the individuals working for them.

Being a small fish, you can chose to swim in the smaller ponds, and avoid the compromises that big business requires. Many of our readers are involved in pocket-sized development projects. I would venture to say that a good number have not built a business to the point where they can leave the corporate world, but they see the promise that one day they will.

Now, why is this a valid topic for a technical journal? First of all, *TCJ* fits the same pattern as its readers. This is a pocket-sized business that I run to avoid the compromises a large publisher would require. It is amusing to get messages asking to speak to our engineering staff or from a foreign university asking me to return the call and "chat." Could you imagine what your subscription would cost if I could afford international calls to chat? If you call *TCJ* and get an answering machine, you now know why.

Custom projects for embedded control are a viable market for a small enterprise. I always enjoy talking to readers (not everyone gets that infernal machine!), and I hear what many of you are up to. Fascinating. The common thread is that these projects are small enough to be manageable by very small businesses, advanced enough to require special skills in both hardware and software and lucrative enough, at least at the small scale presented, to make them viable.

Why would anyone go to an individual to develop something he could have a major house do? Simple answer: Money. Your overhead is a fraction of that faced by the big boys. Anyone needing half a million widgets has the bucks behind him to hire anyone he wants. Many of the projects I speak of are "one-offs" or perhaps a production run of a few dozen.

TCJ is particularly well placed to deal with this. Our authors make frequent use of Forth, an excellent platform for quick development. Several own small companies selling embedded controller boards that one can use to build up a custom system at very low cost. And we have the recognition of some major manufacturers.

You and I have a practical place in this industry; we serve a niche. I hope that you never grow so big that you don't need me, that I never grow so big that I don't hear you, and that we both grow enough to reap modest reward for our efforts.

Serious Hobbyists

Not all readers fit the mold I gave above. Quite a few are what would be considered "serious hobbyists." In his column this issue, Bill Kibler alludes that this entire industry started with games. Perhaps so, but if we go back a bit further, it started with people finding hobbyist use of a new breed of chip—the micro-processor. Those were the days when the 8080 was king, and the 6502 was an up-and-comer. The Altair was the first computer available for home use. It came as a kit and required significant skill to get running.

The early years marked a period when people built their skills to gain functionality as hardware was very expensive. Today, computers are cheap and people have no time to learn. A "power user" is now defined as one who knows tricks in configuring or using software, a microcosm of the total picture.

The computer publishing industry has evolved to meet the needs of the majority. Since few are interested in how a BIOS works, or even that there *is* a BIOS, you will see few articles telling how to write one. Rather than building a capability, most now buy it and so the press emphasizes product reviews. Where does that leave people like you and me?

Tomorrow's "Guru's"

There is one other type of *TCJ* reader: the person who does not currently have the skills to be a "serious hobbyist," but who recognizes that loading another TSR does not represent true skill. This is the same type of person who would rather read a good book than watch the movie. There are no easy paths to knowledge and this person will not take the easy way out. If this means clearing the kitchen table and hauling out the oscilloscope and soldering iron the night *TCJ* arrives, then the kids can have pizza.

If you fit in this category of reader, you understand a

See Editor, page 56

Introducing the YASBEC

Yet Another Single Board Eight-bit Computer

By Wayne Hortensius and Paul Chidley

Once upon a time, there were a couple of computer hackers who were unsatisfied with their computers. One was running a homebrewed 65816 system, and was tired of having to write everything he used himself. The other was running an Ampro LB/Z80, and had finally realized that a single board computer made expansion just a little difficult.

And so, YASBEC was born. It started off as a small project, involving just a few boards. Then came the Trenton Computer Fair, and YASBEC became a unstoppable monster. (By the way Ian, we'll get you for that. Remember that little hassle with Customs? Heh, heh, just you wait. You haven't seen *anything* yet.)

YASBEC is a complete 8-bit, Z180 based single board computer. The footprint of the card is a standard Eurocard, about 4" by 6.3" (for comparison, the Ampro LB/Z80 is 5.75" by 7.75", and the Micromint SB180 is 4" x 7.5"). This sucker is tiny. Like most SBC's, add a power supply, terminal and a disk drive, and you're up and running. But YASBEC goes one step further. It supports a full backplane, to which you can add expansion boards. More about that later.

The Hardware

The hardware that makes up YASBEC is a fascinating blend of leading edge technology, state of the art packaging, and plain old obsolete chips that hardly anyone even makes anymore. Rest assured, there were reasons. Honest. Some of them even still make sense.

The small footprint of the YASBEC board is due to an extensive use of surface mounted components and PALs. Consider; of the 23 ICs on the board, all but 7 are surface mount. The memory decoding is accomplished with a single PAL, as is the I/O decoding.

The heart of the YASBEC is a Z80180 8 bit microprocessor, operating at a clock speed of 9.216MHz. We used the PLCC package, which can address up to 1Meg of memory.

Sockets for both EPROM and RAM are provided on the board. A single 28 pin socket provides for up to 32K of EPROM. This socket can also contain the Dallas SmartWatch, which provides the battery backed up real time clock, at no cost in board space. Very slow EPROMs can be used if required, as the access time can be stretched out to over 400ns.

The system's RAM is provided by two 32 pin sockets. These sockets will accept any of the following static RAM types: 32Kx8, 128Kx8, and 512Kx8. Mapping for the different RAM sizes is controlled by a single PAL. If your RAMs are 100nS or faster, the YASBEC can operate with no RAM memory wait states.

We've been questioned about the decision to use static RAM as opposed to the much cheaper dynamic RAM. Two factors influenced us most: smaller parts count and board space for static RAM, and we had lots and lots of 32Kx8 static RAM's sitting around. Prices on the 128Kx8 static RAM's have also become much more reasonable lately, so that's not as much of a problem as it was. And if you need still another reason, static RAM can easily be made into non-volatile memory with the addition of a Dallas SmartSocket.

And to be honest, both of us hail from a time when the standard joke about dynamic RAM's was, "What's the difference between static and dynamic RAM's? Static RAM works". Old habits die hard.

The Z180 provides two fully programmable serial I/O ports. One channel, intended primarily for a terminal, provides a single flow control input (CTS). The other channel, intended for a modem, provides two flow control inputs (CTS and DCD), and a flow control output (RTS). The RS-232 converter we used is a MAX-239, which happens to provide exactly the right number of inputs and outputs that we needed for the board.

The baud rates for these ports are programmable from 75 baud to 57.6 Kbaud. Any combination of 7 or 8 data bits, 1 or 2 stop bits, odd, even or no parity, can also be programmed.

The singularly useless DCD input can be jumpered to any of three sources: DCD, DSR, or always low. The latter is the most useful option for the vast majority of us.

The parallel printer port provides the ten essential signals of a Centronics compatible printer interface: data bits 1-8, data strobe, and acknowledge. The acknowledge line is tied to one of the Z180's interrupt lines, which makes it very easy to implement a buffered interrupt driven printer port.

A Western Digital 1772 floppy disk controller (FDC) provides all of the functions required to interface with up to four 5.25" and 3.5" floppy disk drives. The 1772 includes the following

Paul Chidley is a senior technologist at NovAtel, an Alberta based cellular phone company. He's a neophyte ZCPR user, but has been active in homebrewed hardware and software design for many years, primarily in the Ohio Scientific and 6502/816 area. Paul can be reached by regular mail at 162 Hunterhorn Drive NE, Calgary Alberta, Canada, T2K 6H5, or by phone at (403)274-8891.

Wayne Hortensius is, in real life, a software designer also, strangely enough, at NovAtel. His involvement with computers began in 1977 when he wirewrapped his first computer together around an 8080A. Wayne's been involved with ZCPR since 1984, on a variety of machines beginning with an Apple II clone and ending up, most recently with the Z180 YASBEC. Wayne can be reached by regular mail at 166 Hunterhorn Drive NE, Calgary Alberta, Canada, T2K 6H5.

functions within a single 28 pin package:

- digital phase locked loop
- digital write precompensation
- motor on start/stop delay
- software controlled step rates from 12ms to 2ms

Timing for the floppy disk interface is derived from an 8MHz crystal oscillator, which is also used to generate the clock required for the optional APU (Arithmetic Processor Unit).

The YASBEC native floppy disk formats are the same as those used on the Ampro LB/Z80 and the Micromint SB180. We could have gone with a newer FDC, and in retrospect, probably should have. The initial decision to go with the 1772 was based on a dream of being compatible with the Ampro LB/Z80. Very early in the design stage, we decided we could do much better than that; but the 1772 had acquired a life of its own.

The YASBEC can be hooked up to a variety of SCSI devices through the NCR 53C80 SCSI bus controller. This low power CMOS device supports all SCSI Initiator and Target functions, including bus arbitration and disconnect/reselect. The YASBEC BIOS uses the SCSI controller in true DMA mode to achieve a very fast data transfer rate.

The YASBEC includes a socket which will accept either the Intel 8231A (a.k.a. AMD9511) or Intel 8232 arithmetic processor units. The socket is jumperable to support both 2MHz and 4MHz parts. The 8231/9511 offers single precision real, and double and single precision integer formats. It performs the full gamut of arithmetic and scientific functions. The 8232 offers single and double precision real formats, but only the basic four arithmetic functions.

While these devices are about as trailing-edge as you can get, having been offered since the mid 1970's, there don't seem to be any other floating point processors that are designed to work with more than one microprocessor. They all seem to be true coprocessors, rather than stand alone devices. [Ed: Is Zilog missing a bet here?]

The backplane is fully buffered, and every signal that you could ever want is out there. Now remember, practically everything you could want in a computer is already on the board, so it's fair to ask exactly what good is having a backplane, when there's nothing to put on it?

Excuse me? The man in the back row? Did you just ask something about graphics? Well, yes, now that you mention it, we have designed a colour graphics board, with two resolutions of 512x424x16 colours, and 256x424x256 colours, with a 16 million colour palette. And yes, it will do frame grabbing (we think). But that's another article, ok?

Also in the works is an upgrade to the system speed. While the YASBEC is already incredibly fast, we're still only running with 10MHz parts at 9.216MHz. Zilog is promising samples of 12.5MHz parts in a month or so, with 16MHz parts by year's end.

The Software

No discussion of a computer would be complete without talking about the software. We're currently running ZCPR3.3 and a modified version of NovaDOS release I (a CP/M 2.2 BDOS replacement) that supports Z80DOS style datestamping. The main reason for this is that is what was running on the Ampro LB/Z80. When all this started last autumn, we had no idea that the YASBEC was going to go beyond a few boards, and didn't put a lot of thought into producing software that could be distributed with a quasi-commercial product. Neither ZCPR3.3 nor NovaDOS is licensed for anything more than personal use, so they were unfortunately not usable. Hey, excuse us, we suffer from terminal honesty. Well, now we've had to think about it, and have decided, just today, to go with ZCPR3.4 and ZSDOS (this ought to make Jay very happy). Many other options were bandied around, but we kept coming back to the problem of having to be able to distribute a bootable system disk. Even a kind offer from Ian Cottrell to distribute the software from his bulletin board couldn't get us around that one.

The BIOS we're using includes many of the features that I'd come to know and love on my Ampro LB/Z80. Things like foreign disk support, automatic sensing of various native disk formats, and an easily configurable disk system. To this add goodies such as: fully buffered, interrupt driven serial I/O, interrupt driven keyboard, an interrupt driven real time clock, and true DMA SCSI transfers. A RAM disk driver is in the works, but you know the old story; so many things to do, and only so many hours in the day.

The boot ROM included with the YASBEC can boot directly from either floppy or SCSI disk. It also includes a monitor for those people who like working with computers without talking to the operating system. As this article is being written, the contents of the boot ROM are in a state of flux, so please excuse us if we're very short on details.

Incidentally, Ian Cottrell is apparently working on getting CP/M+ working on his YASBEC. We're all looking forward

See YASBEC, page 51

YASBEC SPECIFICATIONS

CPU:	9.216MHz Z80180, 8 bit microprocessor capable of addressing 1Meg of memory
MEMORY:	64 Kilobytes - 1 Megabytes of static RAM, with optional battery backup, and up to 32 Kilobytes of EPROM
TIMER:	2 Z80180 Programmable Reload Timers 1 timer not used by YASBEC software
SERIAL I/O:	2 Z80180 Asynchronous Serial Communications Interface channels 2 RS232C compatible ports Software controlled baud rates: 75 to 57.6 Kbaud Modem control signals (DCD, CTS and RTS) on port 0 CTS control signal on port 1 DCD input jumperable to DCD, DSR or permanently low
PARALLEL I/O:	Centronics compatible printer port 10 signals supported Data Bits 1-8 - output Data Strobe - output Acknowledge - input 10 ground pins
SCSI BUS INTERFACE:	SASI compatible ANSI X3T9.2 (SCSI) compatible Uses NCR 53C80 SCSI bus controller
BACKPLANE:	64 pin connector. All signals brought out: power, address, data and control
POWER:	+5VDC and +12VDC
SIZE:	standard Eurocard

Floppy Disk Alignment with the RTXEB and Forth

Third of Three Parts

By Frank C. Sergeant

Frank completes his first place winning entry in the Harris RTX design contest, as he puts all the pieces together for us.—Editor

The Aligner—The Complete System

Tests

1. Sensors

*Write-Protect should go true for a write protected disk and not for a writable disk. *Track0 should go true every time you step to track zero. These statuses (along with the track number and the active head) are displayed on the screen. The words WP? and TRK0? on screen #3605 check the appropriate bit after reading the input port with P@.

2. Speed

The drive should spin the disk at 300 rpm (+/- 6 rpm). A potentiometer at the rear of the drive can be adjusted to make corrections. When the index hole in the diskette is lined up with the hole in the jacket, an LED shines through to activate a sensor, making the *Index line go low. To measure the speed, we set up timer 1 so each roll-over represents a tenth of a millisecond. Then a timer interrupt counts the number of roll-overs. We initialize the counter, wait for the index pulse, unmask the timer interrupt, wait for the next index pulse, mask the timer interrupt, and read the roll-over counter. That is how the word Trev finds the time for one revolution. The square root register isn't needed for any square roots, so it is used as a convenient place to keep the roll-over count. It is initialized to -1 instead of to zero to make up for the 'extra' timer interrupt we almost always get.

SPEED calls Trev and converts it from 10ths of milliseconds per revolution to revolutions per minute and displays the speed. If it is not close enough to 300 rpm, turn the potentiometer until it is.

3. Index to Data Burst

The AAD has a special track with a data burst starting at a fixed position after the index pulse. If the drive's index sensor is positioned correctly, this time will be 200 microseconds +/- 50 microseconds. The word .ITD on screen #3629 measures this by initializing a timer at the start of the index pulse and reading the timer when the first wave peak arrives. This value is in RTX clock cycles. Running at 8 MHz there are 8 clock cycles per microsecond, so the value is divided by 8 and displayed. .ITD uses the word CLOCK to measure the time, even though it is over-kill. See Azimuth below.

4. Azimuth

This is our most difficult measurement. The azimuth track has four data bursts recorded at different head angles. If the head is correctly aligned the middle two bursts will have a greater amplitude than the first and last bursts. Each burst lasts around 400 microseconds, so we have to be quick when we measure the amplitude, and we have to measure it at the correct time. The first thing we do is survey the azimuth track to find when the data bursts occur (in RTX clock cycles relative to the index pulse). This is done by the word CLOCK which stores the timer value when each wave starts and by the word MARK which analyzes the times to mark out the data burst windows. Once we know where to look, AZ measures the amplitudes of the four azimuth bursts. .AZ averages 5 such readings and displays a graph of the relative amplitudes.

5. Cat's Eye

This is the main test. Track 16 is recorded with two out-of-round patterns, so that when the head's radial position (how far it is from the center of the diskette) is correct, there will be two lobes of equal amplitude. The first null point should occur about 25 ms (+/- 5 ms) after the index pulse. The word CE sets up the comparator interrupt and waits for the index pulse, and then waits for the trigger delay after that (this is an adjustable delay-triggered digital 'scope, after all). Then it

Frank Sergeant is a hardware/software consultant specializing in business and/or realtime systems. He is the author/implementor of Pygmy Forth for PC/MS-DOS systems (version 1.3 is available from FIG, GEnie, and fine BBSs and shareware houses everywhere). He has been designing, building, and programming microcomputer systems since the late '70s. One of his greatest joys is replacing hardware with software. He is in the process of porting Pygmy to the Super-8, 68HC11, RTX, etc. His floppy disk drive aligner entry won the RTX design contest. Shortly thereafter he was shocked to hear the RTX was being abandoned by Harris. However, recent conversations with Harris officials have reassured him that it was only future development that was abandoned. Harris has fully, publicly committed to producing the RTX for a minimum of 2-1/2 years. In light of that, Frank breathed a sigh of relief and continues his RTX development work. Frank can be reached as F.SERGEANT on GEnie or through TCJ.

```
scr # 3620
( Vp find peak voltage level of azimuth burst or ce sample)

: Vp ( - dac)
  200 SR!    ( set up the delay value used by ADC)
  ADC  ;     ( measure the voltage)
```

```
scr # 3920
Vp     ("vee pee")
Measures the peak voltage over a short interval. This
is used for both the azimuth bursts and the cat's eye
pattern.

This is our workhorse word to measure wave-form
amplitudes.
```

```
scr # 3621
( AWAIT   Wait for the timer to count down to a specific value)

: AWAIT ( timer1-value -)
  BEGIN     TC1@ OVER U< UNTIL     DROP ;
```

```
scr # 3921
AWAIT      Wait until TIMER1 drops down to the value on
the stack. This is used by AZ so the azimuth bursts
will be measured at the correct times.
```

```
scr # 3622
( Measure azimuth burst amplitudes)

: AZ ( - 4th 3rd 2nd 1st) ( amplitudes of the 4 bursts)
  WINDOWS 12 +  ( a)
  3 FOR ( . . . a)  DUP @ SWAP 4 - NEXT DROP ( t4 t3 t2 t1)
   ( above puts the starting time for each window on the stack)
  SYNC 0 TC1!   ( wait for the index pulse and start the timer)
  3 FOR ( <starting times> )
    AWAIT Vp      ( wait for the next window and take a reading)
    R> SWAP 2>R       ( tuck that reading under the loop index)
  NEXT
  2R> 2R> \\ ;      ( retrieve the 4 readings from return stack)
```

```
scr # 3922
AZ   Measure the amplitudes of the four azimuth bursts.
The WINDOWS array tells it when to do the measuring. It
starts timer1 when the index pulse goes active (SYNC)
and AWAITs the correct timer values before taking the
readings. Note the \\ at the end of the definition.
This forces a separate EXIT instruction. Otherwise, the
compiler seems willing to "optimize" the return with
the 2R> instruction (which it should not do.)
```

```
scr # 3623
( SUM   add corresponding numbers in two sets of four)

: SUM            ( a b c d w x y z   -  a+w  b+x  c+y  d+z)
  4 ROLL + >R    ( a b c w x y)
  3 ROLL + >R    ( a b w x)
  ROT +    >R    ( a w)
  + 2R> R> ;

( this will be used by .AZ to average several sets of readings)
```

```
scr # 3923
SUM  Add corresponding numbers in two groups of 4. This
is used by the following word to average several
azimuth readings.
```

```
scr # 3624
( .AZ  show azimuth test results)
: .AZ ( -)
  !Vb                                    ( find base voltage)
  CLOCK MARK                            ( find azimuth windows)
  AZ AZ AZ AZ AZ           ( take 5 sets of azimuth readings)
  SUM SUM SUM SUM  ( 4th 3rd 2nd 1st)          ( add them up)
  2  ( ie starting-row for cursor position)
  3 FOR         ( make 4 passes, one for each of the 4 bursts)
    DUP 0 AT          ( position cursor on the next row down)
    SWAP 5 / ( we added up 5 readings so now we take average)
    Vb @ -            ( then subtract the base voltage level)
    FOR ASCII X EMIT NEXT  20 SPACES 1+ ( ie bump cursor row)
        ( print horizontal histogram representing amplitude)
  NEXT DROP  ;
: ST-AZ ( -) 34 SEEK   ['] .AZ  HEART ! ;
```

```
scr # 3924
Read Azimuth Amplitudes & Show Results

the peak voltages to something (!Vb). Find when to
sample (CLOCK & MARK). Take 5 samples and average them.
Position the cursor and display a horizontal histogram
with 'X's to represent the amplitudes less the base
voltage Vb.

ST-AZ      Start the azimuth test. Make the azimuth test
the active test.
```

```
scr # 3625
( reduce or divide each sample by a common value )

: LOW-SAMPLE ( - u)  ( find the minimum out of all 200 samples)
  PAD 255 #SAMPLES FOR OVER C@ MIN SWAP 1+ SWAP NEXT
  SWAP DROP ;

: -SAMPLES ( subtrahend -)  SR! PAD  #SAMPLES
  FOR DUP C@ SR@ - OVER C! 1+  NEXT DROP  ;

: /SAMPLES ( divisor -)  SR! PAD #SAMPLES
  FOR DUP C@ SR@ / OVER C!  1+ NEXT DROP ;
```

```
scr # 3925
LOW-SAMPLE Find the group minimum.

-SAMPLES   Subtract a common value from each sample.

/SAMPLES   Divide each sample by a common divisor.

These words are used to scale the measurements for
display on the terminal. (The samples are stored at PAD
by CE.)  These words are called by MASSAGE to massage
the data.
```

collects 200 samples, each a ms apart. (Each revolution takes 200 ms.) MASSAGE massages this data to scale it for display and PLOT draws the 'scope picture as 50 vertical bars of 'X's. The height of each bar represents the average peak amplitude of the sample during that group of 4 ms. With a graphics screen it would be a simple matter to plot all 200 readings (or even more) and have a much prettier cat's eye. This pattern is re-drawn repeatedly. You watch it as you adjust the head position. After the head is aligned, check for excessive hysteresis, or slop, in the head motion. If there is too much slop the head will be in different places depending on whether it was moved inward to track 16 or outward to track 16. To check, move the head to track 34 then back out to 16 and note the appearance of the cat's eye. Then move the head to track 0 and then back in to track 16. If there's too much hysteresis the cat's eye pattern will show the mis-alignment.

User Interface

1. Main Menu

The main menu brings all of these individual tests together and makes it easy for the technician to switch from one to another and to exercise the drive.

The menu continuously runs the selected test and updates the drive status. It responds to single key-presses by moving to different tracks, switching from head 0 to head 1, starting different tests, or changing the 'scope knobs (the trigger point for starting the cat's eye pattern or the scale factor). Where a number is needed, the program prompts for it.

This prompting for a number is a commonly needed function that is often lacking in Forth systems. Here is one way to handle it:

```
: #IN  ( - u)  ( "number in")
  PAD 1+ 5 EXPECT  0 0 PAD CONVERT 2DROP  ;
```

It waits for a number to be typed and then converts it to a number that is left on the stack. Screen #3626 shows examples of how to use it. #IN does not blow up if an invalid or out of bounds number is typed, so it is usually followed by the word CLAMP to keep the values reasonable.

2. Knobs

Two adjustments to the 'scope picture of the cat's eye pattern can be made from the menu. The first is the trigger position. Pressing 'T' prompts for the number of milliseconds delay from the index pulse to the left edge of the display. The second is the range control or scaling factor used to divide the measurements to give a reasonable height display. A good value is 8. This is similar to the volts/division control on a real 'scope. Press 'R' to change the range.

3. Terminal Requirements

The Aligner will run from an XT clone at 115,200 bps. It will start up at any baud rate—it adjusts itself to the speed of the terminal. I keep the source code in BLOCKs and use Pygmy Forth on the XT to edit the source code and download it and to serve as the terminal program in communicating with the RTX board. The source can also be downloaded from a plain text file by an ordinary terminal program. Of course, the code could be put in ROM on the RTX board - there's room for it in the existing EPROM that holds EBFORTH.

The Aligner expects the terminal to be capable of direct cursor positioning and to respond to a clear screen command, and to use XON/XOFF handshaking. Any terminal that can do those three things should work with the Aligner.

4. A Day in the Life of a Technician

The tech opens up the PC's case and removes the drive to be aligned. If the PC has only one drive he temporarily replaces it with a spare he carries for this purpose. Then he connects a power cable extension and Y-adapter to one of the disk power connectors. This powers both the drive being tested and the Aligner. This means the Aligner does not even need its own power supply. He plugs the Aligner's 34 pin data and control cable onto the drive and clips a test lead to the drive's controller board.

That PC (or any handy computer or a portable terminal) provides the interface between the tech and the Aligner through a serial cable. The Aligner hardly even needs an LED on it, never mind a keyboard and readouts! Then the tech exercises the drive through the PC's keyboard and reads the results on the PC's screen as he does the radial alignment, motor speed, track zero, index-to-data, hysteresis, and azimuth adjustments and tests.

5. Enhancements

The question comes up of how fancy we want to plot the measured voltages. The cat's eye routine collects 200 samples per revolution (and could easily collect many more, but what would we do with them?). Then it averages 4 readings into 1 and plots 50 points on the terminal with 'X's. This shows the upper halves of the two lobes. This character graphics approach was taken so it would work with any terminal. The 50 points fit nicely across an 80 column screen. If we wanted to require the terminal to be a PC with a graphics card (or any other graphics environment) or if we added video monitor circuitry to the Aligner, we could plot a very pretty cat's eye display. But, it is very usable with just the 'X's.

If the volume justified it we could put the DAC and the 5 or 6 outputs (we only needed the other 8 because of the DAC) and the 3 or 4 inputs on-chip.

Summary of Advantages

1. Size—Less bulky for on-site alignments. This might make the difference whether the technician is willing to do an on-site alignment. This means faster turn-around for both the customer and the repair shop. "It's ready" rather than "We'll get it back to you eventually" can have far reaching effects on customer relations.

2. Ease of Use—Fewer cables to string, fewer balancing acts to position the drive where it can be worked on. Simple one key control of the head position. Status always visible on screen. All this adds up to a simpler interface and a more complete display than most exercisers provide.

3. Upgradability—Because the essence of this machine is software, upgrades for new and different disk drives can be done with minimal expense: a new ROM and perhaps a new cable. Since the software is written in Forth, upgrades might not even need the ROM. If a PC is used for the terminal, an upgrade diskette with the new Forth commands might suffice. For the technician this means a longer life for his instrumentation investment. For the manufacturer it means the same thing: lower development costs for an upgrade—no new printed circuit board to layout—and the capability (should the impossible happen) of correcting bugs with a ROM change rather than a hardware redesign.

```
scr # 3626
( oscilloscope knobs  )

: OTRIG! ( -) 21 0 AT ." trigger (0-255)? " #IN 0 255 CLAMP
  OTRIG ! 21 0 AT 50 SPACES  ;

: OSCALE! ( -) 21 0 AT ." range (1-255)? " #IN 1 255 CLAMP
  OSCALE ! 21 0 AT 50 SPACES  ;
```

scr # 3926

OTRIG! Lets operator adjust the trigger position for the cat's eye display. ("Moves" lobes left or right.) It is the delayed trigger knob.

OSCALE! Lets operator adjust scaling factor. It is more or less the volts/division knob. Note how CLAMP protects against an entry of zero.

```
scr # 3627
( MASSAGE )

: MASSAGE ( -)
  OSCALE @ /SAMPLES  LOW-SAMPLE  -SAMPLES ;
```

scr # 3927

MASSAGE Divide the raw cat's eye data by the scaling factor and then remove the "DC" component. This helps it fit on the screen.

```
scr # 3628
( Measure drive speed)
: S-INT ( -) SR@ 1+ SR! ;      ( "roll-over" interrupt handler)
: Trev  ( - 10ths-of-a-ms-per-revolution)
  ['] S-INT 8 !INTERRUPT  ( install the roll-over int handler)
  -1 SR!          ( initialize the roll-over counter to "zero")
                          ( the 'extra' int will make it zero)
  SYNC   800 TC1!  TIMER1 UNMASK   SYNC TIMER1 MASK
  SR@   ;

: .SPEED ( -)
  3000 2000 Trev */
  0 <# # ASCII . HOLD # # # #> 19 59 AT TYPE  ;

: ST-SP ( -) ['] .SPEED HEART ! ;
```

scr # 3928

S-INT This timer1 interrupt routine counts the roll-overs (using the SR register, which is available).

Trev ("tee rev") Measure time for one revolution in tenths of a millisecond. The SR register serves as a scratch-pad for counting roll-overs of the timer. Each roll-over represents 1/10th ms as there are 8000 RTX cycles in a full ms. We pre-load SR with -1 instead of 0 as there is "always" a pending timer interrupt when we start.

and display it.

ST-SP Start the speed test. Make the speed test the active test.

```
scr # 3629
( index to data burst timing )

: .ITD ( -)
  !Vb 5 Vb +! CLOCK  ( collect timings for all the peaks just)
                ( so we can use the 1st one - it's over kill)
  PAD 2+ @ NEGATE  ( get the 1st time and convert to positive)
                           ( time in 8ths of a microsecond)
  8 /                            ( convert to microseconds)
  19 50 AT 5 U.R  ;                              ( and display)

: ST-ITD ( -) 1 SEEK  ['] .ITD HEART ! ;
```

scr # 3929

.ITD On the index track of the Analog Alignment Disk (AAD),the data burst should occur 200 usec (+/- 50 usec) after the index pulse. Our wave peak timing collector CLOCK gets this information for us and we read it from the 1st real position at PAD (PAD+2). It is in RTX cycles, which we divide by 8 to convert to microseconds. Then we display it.

ST-ITD Start the Index to Data Burst test. Make this the active test.

```
scr # 3630
( Cat's Eye  - take 200 amplitude readings  - using Vp)

: CE ( -)
  0 Vb !    PAD ( a)
  SYNC OTRIG @ MS                  ( ie wait after index pulse)
  199 FOR  ( a)
    0 TC0!         ( init timer0 so we'll know how long *this* )
                                                ( loop takes)
    Vp Vp + 2/ OVER C! 1+       ( average two readings for this)
                                ( sample and store it at PAD)
    7992 TC0@ + CYCLES     ( kill just the right length of time)
                ( so *this* pass will take 8000 cycles total )
  NEXT DROP   ;
```

scr # 3930

CE Measure the two Cat's Eye lobes per revolution on the CE track (16). Take a reading every millisecond. This is more than we can display on a character-based terminal, but we will average the readings to make everything fit the screen.

Wait for an index pulse. Then wait the trigger value longer, so the operator can control the left/right positioning of the display. The 1st null should occur 25 ms after the index pulse (+/- 5 ms).

Note the use of timer0 to force each pass through the loop to take the same length of time, regardless of the (variable) time consumed by the interrupt handler.

```
scr # 3631
( Cat's Eye)
: AVG ( a # - a' avg)
  0 SWAP DUP >R 1- FOR
  ( a sum) OVER C@ + SWAP 1+ SWAP NEXT ( a' sum) R> / ;

: PLOT ( -)
  11 FOR 13 I - 0 AT  I SR!
  PAD 49 FOR  ( a) 4 AVG SR@ > IF ASCII X ELSE BL THEN
  EMIT  NEXT  DROP NEXT   ;


: .CE ( -)   CE MASSAGE PLOT ;
: ST-CE ( -)  16 SEEK 8 OSCALE !  ['] .CE HEART ! ;

' .CE HEART !
```

scr # 3931

Display Cat's Eye Pattern

AVG Average # samples together. This is used by PLOT.

PLOT For 12 passes, position cursor, store this level in SR (this eases the stack manipulation burden), then go through the data. For each display point, show an 'X' if it is greater than the current level else show a blank. This paints the 'oscilloscope' picture of the cat's eye in low resolution. It still gives a usable representation of a cat's eye. With higher resolution graphics we could show all 200 points (or we could collect even more) and make it pretty. 4 AVG averages 4 samples per display point.

MASSAGE, & PLOT it.

ST-CE Start the Cat's Eye test. Make Cat's Eye the active test.

Summary

Developing with the RTX

The on-board Forth is pleasant with no great surprises. As mentioned before, it makes the use of interrupts very easy.

The RTX is a friendly processor to work with. The Forth development environment and the extreme ease of setting up interrupt handlers make it a pleasure to use. The prototyping board, with the addition of a PC as a host, provides its own development environment. The board wakes up "smart" because of the Forth in ROM. It obeys commands directly from the keyboard or compiles source code downloaded from the host. This entire application fits in the on-board 4K bytes of RAM. The on-board Forth takes up 12K bytes of the 16K byte EPROM, leaving room in the EPROM for the entire Aligner code.

Forth lets you read and set memory, registers, and ports directly from the keyboard. It is easy to toss together little timing and test routines. The feedback is immediate. This gives you a microscope for peering into the hardware so you can see what's going on. I would hate to do without this. If you are beating yourself to death developing hardware using a batch language such as C, Pascal, or assembler, do yourself a favor and try Forth.

Instrumentation products such as the Aligner can expect only a low to moderate volume of sales—far below that of an oscilloscope and far, far below that of a television set or radio. Because of this, there are fewer units to absorb the development costs. We don't want to have to lay out new PC boards whenever an upgrade is desired (of course there won't be any actual bugs to fix). It is much cheaper just to change ROMs. So, for instrumentation work, the RTX, with its quick development Forth environment and the speed to trade for hardware, allows us that less expensive upgrade and maintenance path.

However! When and if the volume justifies it, the per unit cost can be reduced by integrating more hardware functions onto the RTX chip. For example, the DAC, inputs, and outputs could all be put on-chip with the processor. They'd still be on the G-bus, so virtually no change in software would be needed; they'd just be internal devices instead of external. That would eliminate the DAC and three 74HC373 chips, leaving only the external op-amp package (use a dual version that will handle both the signal conditioning and the comparator functions). This would reduce the board size, parts count, and assembly costs.

A designer can get up to speed quickly with the RTX. To start, just lift examples from this and the other published applications. The code for the Aligner shows how simple it can be when done in bite-sized pieces. Speed of development is usually vital to meet windows of opportunity. A fast interactive development environment lets you build a better product. A slow environment encourages you to accept a lower standard because a change is expensive. The RTX2001A lets you replace hardware with speed. The interactive Forth gives speed of development and code compactness and testability that is very hard to achieve with a batch language. This application illustrates these principles in the construction of a floppy drive aligner.●

```
scr #  3632
( Display drive and Aligner status)

: .ST ( -)
 14 23 AT TRK @ 3 U.R
 14 40 AT OUT @ HEAD AND    IF ." 1" ELSE ." 0"    THEN
 14 50 AT TRK0? IF ." TRK0  " ELSE 6 SPACES THEN
           WP? IF ." WP"     ELSE 2 SPACES THEN
 16 40 AT OTRIG @ 3 U.R
 16 56 AT OSCALE @ 3 U.R   ;
```

```
scr #  3932
This is used by the main-loop ALIGN. It shows the
current track and head and whether the write protect
and track zero lines are true. It also shows the
current values for the trigger position and range
(scaling factor).
```

```
scr #  3633
( Display menu)

: MENU  CLS
   0 17 AT ." The Aligner"
  14  0 AT ."  track (-,+,0,1,6,3) =    "
  14 28 AT ."  Head (H) =  "
  16  0 AT ." 'scope (T,R)"
          ."                  Trigger =         Range =    "
  18  0 AT ."  tests (C,A,I,S)"
     ."   Cat's eye    Azimuth    Index to data    Speed" ;
```

```
scr #  3933
MENU      Clear the screen & display the unchanging
text. The changing information is displayed by .ST or
by HEART.

The valid commands are single key-presses, which are
shown in parentheses. Thus '0' moves to track 0 (it
does a RESET); '1' moves to track 1; '6' moves to track
16 (say it "SIXteen"); '3' moves to track 34; 'H'
toggles the head; etc.

The - key is used to exit the menu, back to Forth.
```

```
scr #  3634
( TURN-KNOBS )
: ', ( -) ( input: key action)      ( to help us compile TABLE)
     BL WORD 1+ C@   ,    '  ,  ;

CREATE TABLE      ( to match key-press to an action )
', - -STEP      ', + +STEP     ', 0 RESET     ', 1 TRK1
', 6 TRK16      ', 3 TRK34     ', H -H        ', T OTRIG!
', R OSCALE!    ', C ST-CE     ', A ST-AZ     ', I ST-ITD
', S ST-SP      ', 0 NOP

: TURN-KNOBS ( key -)
  ( look up key in TABLE and perform the matching action)
  >R TABLE BEGIN DUP @ ( a value)
  DUP R@ = SWAP 0= OR 0= WHILE ( a) 4 + REPEAT ( a)
  DUP @ R> = IF ( a) 2+ @ EXECUTE ELSE DROP THEN  ;
```

```
scr #  3934
,'   "comma-tick" reads the input stream and 'comma's
in the key and its corresponding action. This helps us
build the following table.

TABLE      This is used by TURN-KNOBS to match an action
to a key stroke. The final entry must be for 'key'
zero. Note that with the help of ,' we do not have to
say ASCII C  to indicate the "C" key. It simplifies the
source code that creates this table.

TURN-KNOBS      Look up key in TABLE and perform the
corresponding action. If no match, do nothing.
```

```
scr # 3635
( ALIGN    top level word disk aligner)

: ALIGN ( -)
  RESET
  BEGIN MENU .ST
       BEGIN
          19 ( XON) EMIT            ( tell terminal we're busy)
          HEART @ EXECUTE           ( perform the current test)
          17 ( XOFF) EMIT                  ( let terminal talk)
          100 MS           ( give terminal time to send a char)
       KEY? UNTIL          ( keep going until a key is pressed)
       KEY  DUP 126 -  WHILE          ( if not the ~ key, then)
          BEGIN TURN-KNOBS KEY? WHILE KEY REPEAT    ( obey the)
  REPEAT DROP ;                          ( command and continue)

( ****************          The End          ****************** )
```

scr # 3935

ALIGN Where It All Comes Together. Display menu and status of drive and program. Start with the Cat's Eye test. Keep displaying the active test (whose execution address is in HEART) until operator changes the active test. Change tests, head, track, trigger position, range (scaling factor) per operator's single keystroke commands.

```
scr # 3636
( flash LED in time with index pulse - it's comforting )
VARIABLE OUT   ( so we can remember how we set output port )
: ON  ( bit-mask -)     OUT @  OR DUP OUT ! 31 GI ;  -1 ON
: OFF ( bit-mask -) NOT OUT @ AND DUP OUT ! 31 GI ;
  ( set or reset a bit without disturbing the others)

1 CONSTANT  LED          ( output port bit mask for LED )
1 CONSTANT *INDEX        ( input port bit mask for index pulse )

: P@ ( - u) 31 G@ ;      ( read input port )
: INX?  ( u - f) *INDEX AND 0= ;  ( true if index pulse active)

: SHOW-PULSE ( -)
  BEGIN KEY? 0= WHILE P@ INX? IF LED ON ELSE LED OFF THEN
  REPEAT  KEY DROP ;
```

scr # 3936

Get Some Feedback: It's comforting and gives you the courage to continue.

The output port's LSBit is connected to an LED. You can make it go on with 1 ON and off with 1 OFF. Try it. Now you know you can set an output bit. The input port's LSBit is connected to the disk drive connector's *index line. The asterisk means it is active low (the line reads as zero when the index hole in the diskette is aligned with the hole in the disk jacket and light shines thru both holes. Otherwise that line is high.

Read the *index line and turn on the LED when the *index line is low and turn off the LED when the *index line is not low. The LED should flash 5 times per second when the disk is spinning.

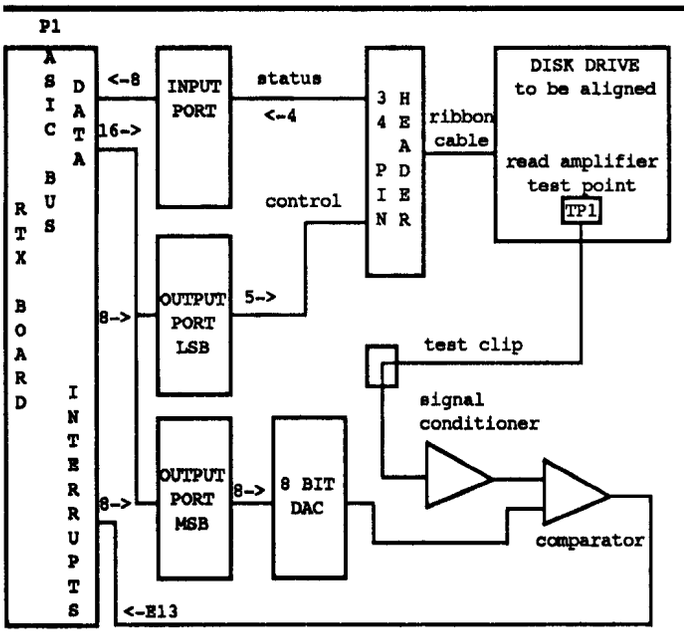
```
scr # 3637
( T  .T    find execution time of a section of code )

: T  ( -)   0 TC1!  ;  ( initialize counter)

: .T ( -)
  TC1@  NEGATE  1-       ( make timer value positive and adjust)
  -1 IMR!                              ( mask all interrupts)
  EI5 UNMASK                     ( then unmask the one used )
                                       ( for the serial line)
  U. ." cycles " ABORT  ;          ( print result as we abort)

: X1  T  .T  ;                ( it better be zero)
: X2  T NOP .T ;              ( it better be one)
: X3  3  T  FOR NEXT .T   ;   ( it better be five)
```

scr # 3937

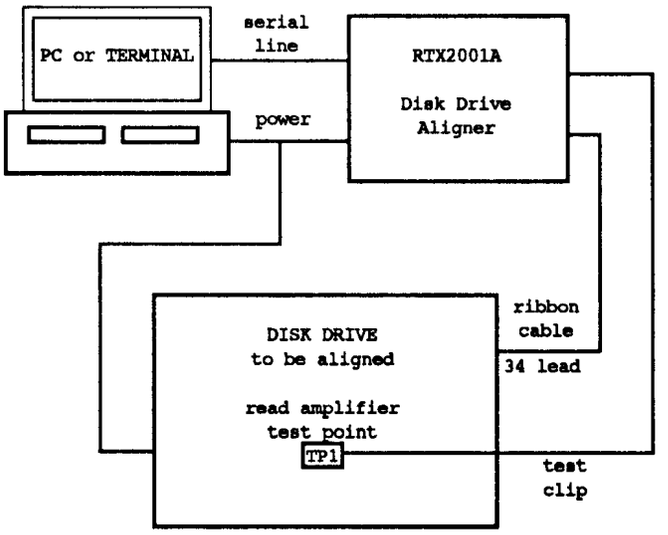
Find Execution Time of a Section of Code

This is a very handy DIAGNOSTIC aid. Don't guess, don't add up instruction timings, don't guess what the optimizer did. Instead, slap a T before the troublesome section and a .T after it and run it. The EXACT number of cycles the sequence takes will be displayed!

T & .T were used during the testing of CE to determine the exact value to subtract the timer from to get a fixed length loop regardless of the time spent in the interrupt handler. It was also used in several places to make sure time constraints were met and to satisfy curiosity about certain instruction timings.



HARDWARE DIAGRAM



OVERALL SYSTEM CONNECTIONS

High Speed Modems on Eight-Bit Systems

This is Not Your Father's Ampro

By Roger Warren

Introduction

Once upon a time, I had a job that, in part, involved placing a company's computer system on-line so that clients could call in and use some the company's proprietary programs. It was an interesting time. I was *very* green. I had nearly barely heard of a modem before that time and, worse yet, the minicomputer I was using had no system software packages to support such an operation, so I had to learn about telecomputing as I developed the software. My operator's console was a Teletype model 33 operating at 110 bits per second. At the time, modem signaling rates in excess of 300 baud were exotic. I learned all about flow control: the machine could deliver data to the modem and Teletype *much* faster than either could process. The RS-232 Request to Send and Clear to Send signals (RTS and CTS) were employed for the modem. Similar controls were used for the Teletype.

After that job, I conveniently ignored flow control. It just wasn't necessary on a daily basis. CRTs seemed to be able to gobble data as fast as it could be delivered. Some didn't bother to support flow control! In general, a three wire subset of the RS-232 standard seemed to be sufficient for most things.

Recently, I decided to outfit my Z-Node with a USR Courier HST Dual Standard modem. The real truth of the matter was that I was finally just fed up with the antics of my previous 2400 baud modem—it had been performing great feats of random firmware amnesia for years, and I was tired of modifying my system's BYE program to accommodate each newly-discovered malady. I had reached the point where I was willing to shell out the dollars to buy freedom from the beast, but with an eye toward the future, too.

My flow control experience came in handy.

Most of us who operate 8-bit machines have not really run into any speed-related problems with RS-232 communications. The system CRT always seems to handle data from the CPU at whatever baud rate is being used, and fingers can't type at a rate approaching that at which the CPU can accept and process data. However, there *is* a performance envelope to deal with. For a Z80 operating at 4 MHz, approximately 650 instructions (using an average of 7 clock cycles per instruction) can be executed in the time it takes for a byte to be sent or received over a 9600 baud serial link. This may *sound* like a lot, but, really, it's not! When your modem program sends or receives a byte from your modem, several hundreds of instructions are usually involved just in dealing with the modem. Add to that the facts that your console input must be polled for operator keystrokes (requiring more instructions) and that some kind of console display update will probably be required (still more instructions), and it becomes obvious that there's a potential that the CPU may not be able to keep up with a 9600 baud data stream.

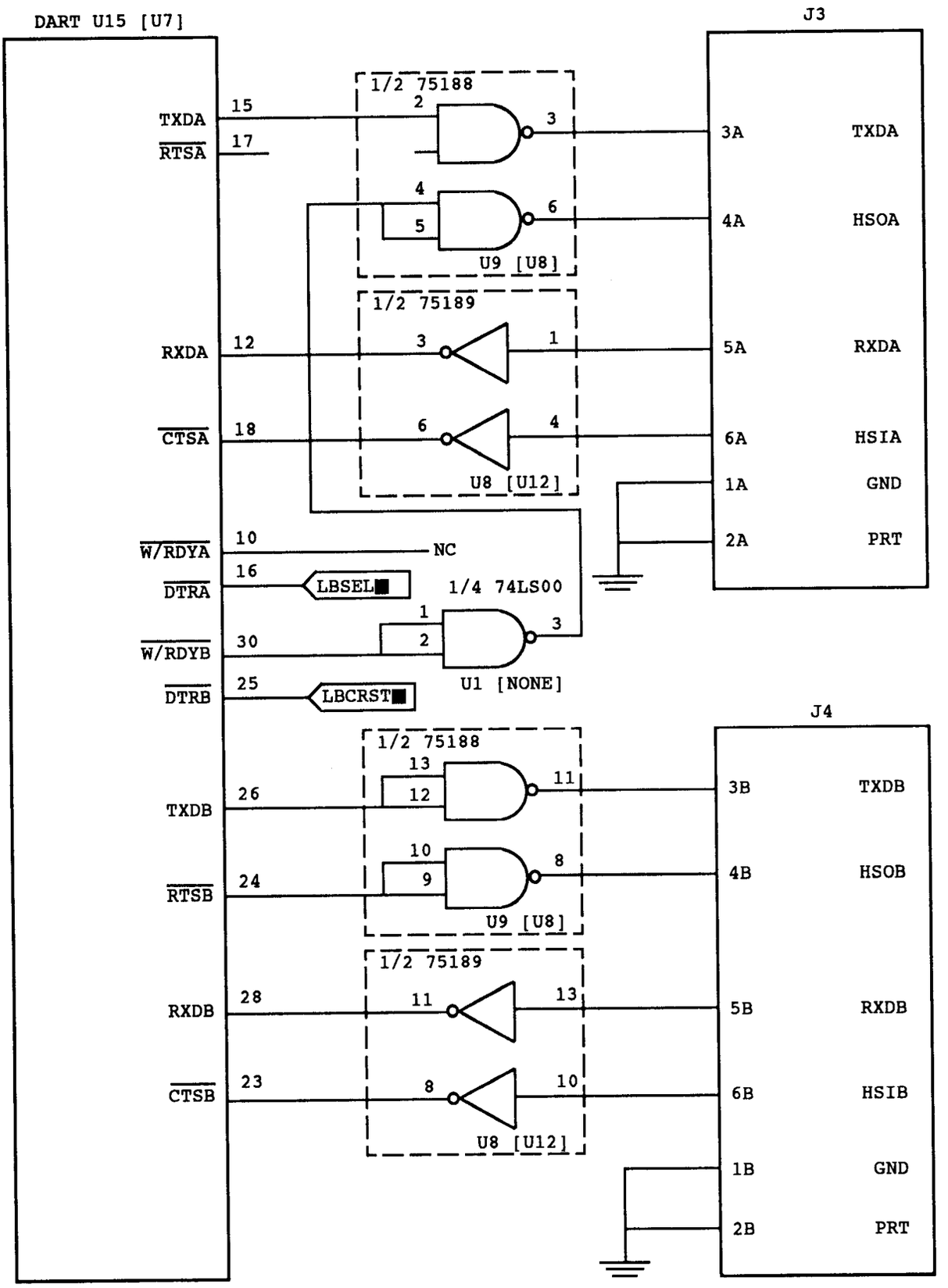
When I first attached the new USR to my old Ampro (the 4 MHz Z80 that runs my Z-Node), I quickly found that the 'effective' maximum baud rate for my configuration was between 4800 and 9600. That is, things seemed to work well at 4800 baud and lower, but when I used 9600 baud or higher the CPU couldn't keep up with the modem, causing data to be lost. The error correction feature of the modem made things worse: when used, data is sent to the CPU from a buffer in the modem at whatever rate it can pump data out—not at the rate the computer on the other end of the modem connection is supplying the data. Flow control was required keep the CPU from losing information!

For the benefit of those who don't have a clue as to what flow control as implemented on the USR may be, let me provide a brief summary: When the computer is able to accept data, it asserts a signal to the modem. When it's unable to accept data—like when the CPU hasn't read a previously received character—the computer removes the signal. This is the receiver flow control. On the transmitter side, the modem sends a signal to the computer when it's able to accept data and removes the signal when it's unable to accept data. USR uses the RS-232 signals Request to Send (RTS) for receiver flow control and Clear to Send (CTS) for transmitter flow control.

Before going further, it's worth observing that USR's use of RTS for receiver flow control, while workable, is contrary to it's traditional use. Originally, when modems were much slower than computers, the computer asserted RTS when it wanted to transmit a character and the modem responded with CTS when it could accept the character. There was no corresponding handshake when the modem

Roger Warren is former metallurgical engineer who has been making a living as a system, software, and hardware engineer since being introduced to computing in the early 70's. He's been in the aerospace and defense fields for a majority of that time, working primarily with embedded controls and radiation-hardened memory systems.

He's been active in the CP/M and ZCPR communities for several years, having developed and released several enhancements to the Ampro BIOS and, more recently, LZH compression for CP/M (CRLZH and UNCRLZH). He's been operating Z-Node #9 (aka 'The Elephant's Graveyard') in San Diego since 1986. Those who'd like to see that the USR/Ampro combination in action can call the Elephant at 619-270-3148. (PCP: CASAD, SL: 9183)


DART U15 [U7]
J3
TXDA
RTSA
RXDA
CTSA
W/RDYA
DTRA
W/RDYB
DTRB
TXDB
RTSB
RXDB
CTSB
15
17
12
18
10
16
30
25
26
24
28
23
1/2 75188
U9 [U8]
1/2 75189
U8 [U12]
NC
LBSEL
1/4 74LS00
U1 [NONE]
LBCRST
3A
TXDA
4A
HSOA
5A
RXDA
6A
HSIA
1A
GND
2A
PRT
J4
3B
TXDB
4B
HSOB
5B
RXDB
6B
HSIB
1B
GND
2B
PRT

```
Listing 1
DART initialization modification for AMPRO BYE insert


PORT    EQU   88H       ;Data port
MDCTL1  EQU   PORT+4    ;Status/control port
;
; Disconnect and wait for an incoming call
;
MDINIT: MVI   A,18H     ;Reset channel
        OUT   MDCTL1
        MVI   A,4       ;Setup to write register 4
        OUT   MDCTL1
        MVI   A,8CH     ;*2 stop, 8 bits, no parity, 32x
        OUT   MDCTL1
        MVI   A,5       ;Setup to write register 5
        OUT   MDCTL1
        MVI   A,68H     ;Clear RTS causing hangup
        OUT   MDCTL1

; 11100000B to DART/SIO
; Enables W/RDY as a DMA RDY rqst for Receiver
        mvi   a,1       ;set up for reg 1
        out   mdctl1
        mvi   a,0E0h    ;Enable RDY (for RTS)
        out   mdctl1

        PUSH  B         ;Save in case it's being used
                        ;elsewhere
        MVI   B,20      ;2 seconds delay to drop any carrier
```

wanted to send data to the computer—the computer, being faster, could handle whatever the modem sent. While this shift in functionality of the RTS line from request for the computer to send data to the modem to request for the modem to send data to the computer makes sense under the circumstances, it may not be embraced by most existing communication hardware.

My point in bringing this up is this: One could not just hook up the USR to an older machine, even if the machine had support for the RS-232 RTS and CTS pair, and expect flow control to work. Serial communication chips which interface with these signal are designed to support the traditional use. It's likely that most of the other newer, higher speed modems will have similar interfacing differences, so be sure to do a bit of research before buying any new modem.

The remainder of this article describes the details of hardware and software changes required to add a version of hardware flow control on the Ampro Z80 Little Board so that the USR can be used with the machine at 9600 baud. The decisions I made when designing those changes are also presented. Those of you who don't own Ampros should not just stop reading at this point. The Ampro uses either the Z80A DART or the Z80A SIO/0 (they're interchangeable in the application) for serial communication, which are very commonly used on a variety of 8-bitters. While what follows is Ampro-specific, it will be generally applicable to many other machines.

Deciding How Far to Go

Compatibility was a major consideration when I was deciding how to interface with the USR. I did not want to make existing Ampro overlays for MEX, IMP, BYE, et cetera, unusable on the machine. I also wanted to be able to use one of my spare (slower) modems without major grief—and to not have to maintain two sets of modem programs to do so.

First, I decided not to bother with the USR's 12,000 and 14,400 baud rates. The Ampro's serial port B, with a maximum baud rate of 9600, is the 'standard' modem connection. Since the Ampro's 'effective' maximum baud rate is about 4800 (somewhat greater when doing file transfers because there's less screen updating going on), 9600 is optimal. Happily, the USR can be commanded not to use the higher rates, and MEX and IMP overlays for the Ampro already support 9600 baud, so there seemed to be no real good reason to make any modifications to support the USR's higher rates. Yes, I could have modified the Ampro to support higher rates on serial port B, or could have operated the modem on serial port A (which will go as high as 38400 baud), but either approach would require more overlay modifications than I was willing to accept.

While it was immediately obvious that flow control was required for the receiver (my IMP program couldn't even field the USR's HELP screens without losing data), the need for transmitter flow control was not as readily obvious.

Under conditions of poor line quality, the USR (if connected with another modem of similar capabilities) will 'fall back' to a lower baud rate. When this happens, the communication rate with the computer is not modified, and data is supplied to the modem faster than it is sent out. The modem buffers the host's data, but flow control is required to keep data from being lost if the buffer becomes full.

If error correction is in use and the receiving modem detects an error, the transmitting modem responds by resending a block of data. While the block is being retransmitted, data from the host is being buffered. Again, flow control is required to keep data from being lost.

Having determined that both receiver and transmitter hardware flow control were required (software flow control was never a serious consideration), the means to provide the control had to be found.

```
Listing 2:
Output status routine modification for AMPRO BYE insert


;
; See if the output is ready for another character
; Return ZERO (w/flag set) if not ready
;
mdctl2  equ   80h+4     ;Secondary control port (modem CTS)

MDOUTST:IN    MDCTL1
        ANI   04H       ;Check transmit ready bit
        rz              ;No data can be sent
        mvi   a,10h     ;Reset Ext status data
        out   MDCTL2
        in    MDCTL2    ;Read CTS data
        ani   20h       ;Check CTS
        RET
```

Working With What's There

While the folks at Ampro have really squeezed a lot of functionality out of the small number of chips on their board, there's not a whole lot of extra stuff on the board to play with. My changes have been managed by stealing (generally unused) parts from the interface to the operator's console. However, I couldn't avoid the need to add one IC (an inversion of a logic signal was required, and there were no spare gates to steal). More on that later. A cut and a few jumpers are also required, but more on that later, too.

First, the hardware to provide the flow control needed to be found. Ampro has provided both serial ports with a pair of handshaking signals: HSOA (output) and HSIA (input) for port A, HSOB and HSIB for port B. They are intended to provide limited support for flow control. There's BIOS sup-

port for the use of these signals, which *must* be disabled (a typical configuration) for the modifications described here to work. The HSOB and HSIB signals of the modem port had already been appropriated by communication program overlays for use as the Data Terminal Ready (DTR) and Data Carrier Detect (DCD) signals, respectively. The HSOA and HSIA lines from the system console (serial port A) weren't being used, so their connector pins, RS-232 transmitter, and RS-232 receiver could be borrowed. These would be interfaced to the modem's RTS and CTS signals, respectively.

Next, the means to interface these lines to the serial port B needed to be found. The modem's CTS (HSIA) was simple—I *didn't* interface it to serial port B. I left it where it was! Not only had the real CTS function on serial port B been usurped for DCD, but the other inputs related to serial port B, DCD (the pin on the DART, not the DCD from the modem) and Ring Indicator (RI), were already used by Ampro for disk and printer ready status information. Leaving HSIA (the modem's CTS) as an input on serial port A did not impact port A operation. Four instructions are required to sense its state when a MEX overlay or BYE insert requires the information. Best of all is the fact that unmodified communication program inserts, lacking those additional instructions, would operate fine with the USR at 2400 baud (or with other 2400 baud modems)!

The interface for the modem's RTS line was not as easy. There were no spare output signals (none under direct software control, that is) on either serial port. True, the DTR output for serial port B is assigned as an (optional) floppy disk controller reset control on the Ampro CPU 1B version (it's a spare on the CPU 1A version), but I decided against employing it. My major reason for rejecting it was the fact that serial port B's RTS control (used by modem inserts for the DTR function) shares a control register with the DTR control. Since these controls would need to be operated independently by separate pieces of code, some coordination between those routines would be required. The software changes to inserts and overlays to accomplish the coordination wouldn't have been massive, but they were more than I cared to accept. Besides, I had an idea that appealed to me more.

The Z80 DART (and SIO/0) have provisions to operate with a Direct Memory Access controller (DMA). These provisions aren't used on the Ampro. When the feature is used, the Wait/Ready output of a DART channel is connected to the Ready input of a DMA. When the output is asserted, the DART is signaling the DMA that it is ready to transfer data. This operation was perfectly suited to providing the necessary control of the USR's RTS input: The signal is in one logic state when the DART's receiver is empty (is able to accept data) and in the other logic state when the DART's receiver is full (and can't accept data—like when software hasn't read previously received characters from the DART's data register). An added plus was that, aside from the addition of a few instructions to the initialization portions of communication program inserts and overlays (to program the DART to use the DMA request signal on received data), there would be no software impact.

There was a small problem with using this signal: it's logic state is reversed from what was required. When used as a DMA receiver ready control, the W/RDYB* signal (the '*' is the ASCII text equivalent for the bar drawn over the signal name) is driven low to signal the DMA that it has a byte ready to transfer (receiver full), and high when no data is available (receiver empty). The RTS* signal, on the other hand, should be low to indicate a receiver ready (empty) condition, and high for a receiver not ready (full) condition. The addition of an inversion was required.

```
Listing 3:
BAUD rate selection expansion for AMPRO BYE insert

;
; The following routine sets the baudrate.  BYE5 asks for
; the maximum speed you have available.
;
SETINV: ORI   0FFH      ;Make sure zero flag is not set
        RET             ;Return
;.....
;
;
SET300: LXI   H,BD300   ;Get 300 bps parameters in 'HL'
        JMP   LOADBD    ;Go load them
;
SET1200:LXI   H,BD1200
        JMP   LOADBD
;
SET2400:LXI   H,BD2400
        jmp   loadbd
;
set4800:lxi   h,bd4800
        jmp   loadbd
;
set9600:lxi   h,bd9600
;
LOADBD: mvi   a,4       ;Sel reg 4
        out   mdctl1    ;like so
;
        mov   a,m       ;get data for clock etc.
        inx   h         ;goose pointer
        out   mdctl1    ;to port
;
        mov   a,m       ;get ctc1 values
        inx   h         ;goose pointer
        OUT   BRPORT
        mov   a,m       ;get second part
        OUT   BRPORT
        XRA   A
        RET
;.....
;
-------------------------------------------------------------
Locate this next part with the constants between the
labels ENDOBJ and PEND
-------------------------------------------------------------
;
; TABLE OF BAUD RATE PARAMETERS
;
bd300   db   8ch,47h,208 ;data reg 4,ctc command,ctc value
bd1200  db   4ch,47h,104 ;data reg 4,ctc command,ctc value
bd2400  db   4ch,47h,52  ;data reg 4,ctc command,ctc value
bd4800  db   4ch,47h,26  ;data reg 4,ctc command,ctc value
bd9600  db   4ch,47h,13  ;data reg 4,ctc command,ctc value
```

As I stated previously, there's not a whole lot of extra stuff to play with on the Ampro—and a spare inverter was not to be found. Granted, I could have stolen the necessary logic function from somewhere else on the board, but something would have had to have been sacrificed to do so. The addition of a chip seemed best. The Ampro CPU 1B has a spare 14-pin IC location on the board, and this was the proper occasion for its use. The Ampro CPU 1A (I own both types) does not have a spare location on the board, so adding a chip on that board required other methods (I chose to piggy-back the added chip onto an existing 14-pin IC).

Modifying the Board

The modification to the Ampro board, itself, is small. I realize that some really won't feel comfortable with or ca-

See Ampro, page 52

A Z8 Talker and Host

By Brad Rodriguez

Introduction

What do you do with a new, unfamiliar CPU, with no accompanying development tools? Perhaps a board of your own design, with untested logic? You need a program which will let you both exercise your hardware, and load and debug software. You need a "debug monitor."

But...without a debug monitor to debug your debug monitor, this program should be as utterly simple and obvious as possible. Something you can get running with nothing more than a PROM burner.

You'd need at least three capabilities: examine the target system's memory, alter its memory, and start a program at a given address.

In Forth circles, such a rudimentary monitor is known as a "talker" program. This article describes a talker program for the Zilog Z8. In addition to the basic features, it has embellishments such as register access and a breakpoint facility. The talker is small (under 300 bytes), easy to port to new processors, and easy to get running. I've used it to debug wirewrap prototypes, to bring up Forth kernels, to develop and debug large assembly language programs, and to debug applications in the field.

The Talker Program

The secret to keeping the talker simple is to offload most of the work to a host computer. So, there are really two programs involved. The "talker" program runs on the target hardware, and communicates via a serial port with a "host" program running in a personal computer. I'll discuss the talker program first.

History and Design Philosophy

I originally wrote the talker program to replace Zilog's debug monitor for the Super8. Zilog's monitor program had several shortcomings:

1. It was too big to be embedded in an application program—it occupied 12K of ROM, needed 1K of RAM, and used many of the CPU registers.
2. Because it needed RAM, it couldn't debug "RAM-less" hardware (or the RAM memory decoding).
3. Because it took over UART interrupts, and shut off others, it couldn't debug interrupt routines.
4. It couldn't be "polled" in the background while other tasks were running.
5. Its access to Super8 registers was flaky.
6. During breakpoints, it was difficult to examine the processor state.
7. It couldn't run on a half-duplex serial line (which is what much of our hardware had).

I wanted a program which was more useful for debugging tricky Super8 code, and which I could embed in a final application. The program should:

1. use a minimum of ROM
2. use a minimum of on-chip RAM (register file)
3. use NO off-chip RAM, if possible
4. use NO off-chip I/O, if possible
5. use NO interrupts
6. use half-duplex serial communications

The simpler Z8 version described here uses only 300 bytes of ROM, 6 bytes of on-chip RAM, and the on-chip UART. The half-duplex feature is disabled, but can easily be re-installed by editing two macros.

Basic Operation

Figure 1 is the listing of the Z8 talker program.

The program, TALKER, repeatedly calls the routine TALK to poll the UART for received characters, and to process them when received. TALK was "factored out" as a separate subroutine so that it could be called periodically from an application program. (Obviously we can't call TALKER, an infinite loop, while an application is running.)

A received character is either a command, or a hex digit. To simplify conversion, the characters 0123456789:;<=>? (hex 30-3F) are used as the hex digits.

The talker program maintains 4 "virtual registers":

MDR: memory data register, 1 byte

MAR: memory address register, 2 bytes

MBR: memory bank (page) register, 1 byte

WKG: working (scratch) register, 1 byte

(A sixth byte is used to hold temporarily the contents of the Z8's RP regis-

Brad Rodriguez, of T-Recursive Technology, Toronto, is a freelance software/hardware designer specializing in real-time control applications. He has been working with the Zilog Super8 since 1987, and is the proud father of several Z8 and Super8 Forth kernels. Forth has been his language of choice since he discovered it in 1978, much to the dismay of his friends, co-workers, superiors. His thesis advisor shares this dismay, because Brad is currently working part-time toward a PhD in Electrical Engineering, in the field of real-time Artificial Intelligence. In his Copious Free Time he pursues anachronistic endeavors such as building a Forth processor out of TTL logic, building an 8-bit multiprocessor system, and writing assembly language code. His ambition of the moment is to see Forth displace LISP completely. Brad can be reached as B.RODRIGUEZ2 on GEnie, or as bradford@maccs.dcss.mcmaster.ca on the Internet. He prefers the former.

ter.)

When a hex digit is received from the host program, it is shifted into the low 4 bits of MDR. The previous low nybble is shifted up, and the previous high nybble is lost. Thus, sending "23" from the host will set the MDR to 23 hex.

The characters 20-2F hex are used for the commands. The basic command set has these six commands:

```
HEX ASCII  COMMAND
2A   *     fetch byte from MAR address
           into MDR, send to host, and
           increment MAR
2B   +     store byte from MDR into MAR
           address, and increment MAR
2C   ,     copy MDR to low byte of MAR
2D   -     copy MDR to high byte of MAR
2E   .     copy MDR to MBR (memory page)
2F   /     start program at MAR address
```

The memory address and page are set by loading a byte into the MDR, then transferring it to the desired register. So, to specify memory address 1234 in page 00, the host sends the nine characters

```
0 0 . 1 2 - 3 4 ,
```

The MDR is then free for data. The host can send two more digits and then a "+" command to write data to memory. Or, the host can send a "*" command to read data from memory. In this case, the talker program will send the data byte as two hex digits, using the characters 30-3F hex (as above).

Note that the talker program *only* transmits on request from the host. In a half-duplex environment, the target will turn on its transmitter, send the two digits, and turn off its transmitter. The host, of course, knows to turn off its transmitter until two digits have been received.

The fetch and store operators autoincrement the address to make downloads, uploads, and memory dumps more efficient. For example, you can dump 16 bytes starting at address 1234 with the command sequence

```
1 2 - 3 4 , ****************
```

The "memory page" register was included to allow access to multiple address spaces. The Z8 has 64K of "Code" memory, 64K of "External" memory, and 256 bytes of on-chip RAM (register file), selected by page numbers 0, 1, and 2, respectively. This mechanism could also be used for I/O space, bank-switched memory, or ex-

```
;  FIGURE 1.  THE TALKER PROGRAM.

;=====================================================================
;
;       ZILOG Z8 "MICRO-TALKER" MONITOR PROGRAM
;       (c) 1990 T-Recursive Technology
;       placed into the public domain for free and unrestricted use
;
;       A minimal monitor program for the Zilog Z8.
;       Used in conjunction with a "smart" host program to examine &
;       modify code memory, external memory, and registers, and
;       to set and execute breakpoints in machine language and Forth.
;
;       The Talker program uses only 6 registers (0Ah to 0Fh here),
;       no RAM, and about 300 bytes of PROM.  No interrupts are used.
;       The Talker may be operated half-duplex over a bidirectional
;       serial line.
;
;       The program accepts characters from the Z8 UART.
;       Characters 30h to 3Fh are treated as hex digits and are
;       shifted into a one-byte data register.
;       Characters 20h to 2Fh are command codes:
;           20-23 reserved for future use (ignored)
;         group 2: breakpoint / debug support
;           24 = set a Forth "thread" breakpoint
;           25 = set a Forth "code field" breakpoint
;           26 = set a machine language breakpoint
;           27 = fetch lo byte of address (adrs lo -> data & output)
;           28 = fetch hi byte of address (adrs hi -> data & output)
;           29 = read back data register  (data -> output)
;        group 1: minimal talker
;           2A = fetch byte & incr addr   (mem or reg -> data & output)
;           2B = store byte & incr addr   (data -> mem or reg)
;           2C = set lo byte of address   (data -> adrs lo)
;           2D = set hi byte of address   (data -> adrs hi)
;           2E = set memory "page"        (data -> page)
;           2F = "go"                     (jump to current adrs)
;
;       The memory "page" is interpreted as follows for store and fetch:
;           0 = C (code) memory
;           1 = E (external) memory
;           2 = registers 00-FF
;
;       Internal register usage:
;           rr14 (0E,0F) = address
;           r13  (0D)    = memory "page"
;           r12  (0C)    = data byte
;           r11  (0B)    = working
;           r10  (0A)    = saved rp1
;
;       Revision history
;       v 1.0  23 Aug 89    original Super8 program; functions 29h - 2Fh;
;              access to memory, registers, indirect registers;
;              'talker' and 'talk' entry points.
;
;       v 1.1  25 Feb 90    support for breakpoints and Forth words;
;              standalone initialization.
;
;       v 1.2  7 May 90     function codes reassigned; added address
;              readback; improved breakpoint support; fixed problem
;              with direct register access to C8-CF and to RP1...now
;              correctly uses application program's registers.
;
;       v 1.2CP  7 Jun 90   modified for Teatronics Echelon Channel Proc,
;              as an include file.  Uses half-duplex RS-485.  Sets P37
;              low to turn LED 'on'.  Pulses watchdog on TXD.
;              Fixed bug where tx turned off during last tx character.
;              Changed to disable irpts on 'mbreak' entry, enable on 'go'.
;
;              26 Jun 90    altered to assemble standalone; set IMR to 00
;              so that irpt enable on 'go' doesn't lock up system.
;
;       Z8 v1.0 2 Dec 90  modified for Zilog Z8
;
;=====================================================================
;
;       Monitor Configuration and Assembly-Time Options
;
;       STANDALONE - set to '1' if this file is to be assembled as a
;           standalone program, to be put into PROM.  Set to '0' if
```

```
;          this file is to be 'included' in another source file.
;
standalone .equ 1

;
;       Macros for half-duplex communication on a bidirectional link,
;       e.g., a bidirectional RS-485 serial line.  Define these macros
;       to control the transceiver connected to the Z8's serial port.
;       If full-duplex is to be used (separate transmit and receive data
;       lines), define these as "null" macros.
;
;       TXON - turns serial port transmitter on, and receiver off.
;       TXOFF - turns serial port transmitter off, and receiver on.
;       KICK - kick the watchdog timer (v. 1.2CP)
;
txon    .macro
        .endm

txoff   .macro
        .endm

kick    .macro
        .endm

;
;       Register bank used by monitor: the monitor requires 6 bytes
;       of Z8 registers.  These must be the last 6 bytes of a
;       16-byte register bank, since they will be mapped (via RP) onto
;       working registers R10-R15.  These will be accessed directly
;       as working registers AND as general purpose registers.
;       Set 'regs' to an 16-byte boundary between 00 and 70, inclusive.
;
regs:   .equ    00h     ; monitor will use regs+0b to regs+0f (0B to 0F)

rplsv:  .equ    regs+8+2  ; save area for applic's rp
wkg:    .equ    regs+8+3  ; working "scratch" register    aka r11
mdr:    .equ    regs+8+4  ; memory data register          aka r12
mbr:    .equ    regs+8+5  ; memory bank register          aka r13
mar:    .equ    regs+8+6  ; memory address register pair  aka rr14

;=============================================================================
;
;       STANDALONE INITIALIZATION
;
;       For use when the monitor is used as a standalone program
;       in a Z8 development board.  In this case, the monitor
;       is located in low PROM, to be started on reset.  The Z8
;       registers are "minimally" initialized, to allow full-duplex
;       serial communication at 4800/9600 baud, 8 bits, no parity.
;       The interrupts are vectored to a supplementary jump table
;       in code RAM.
;
;       Expects:  reset state for all Super8 mode & control registers
;       Returns:
;       Uses:
;
;=============================================================================
        .if standalone
;
;       interrupt vectors
;
vecs    .equ    0e000h  ; base address of the interrupt jump table

        .org    0       ; the Z8 interrupt vector table
        .word   vecs    ; irq0  p32
        .word   vecs+3  ; irq1  p33
        .word   vecs+6  ; irq2  p31, Tin
        .word   vecs+9  ; irq3  p30, serial in
        .word   vecs+12 ; irq4  T0, serial out
        .word   vecs+15 ; irq5  T1
;
;       z8 initialization...immediately follows vectors
;
        clr     p0              ; output 0 hi adrs bits, just in case
        ld      p01m,#10110010b ; p0=a8-a15, p1=ad0-ad7, ext'l stack, slow mem

        clr     p3              ; output 0 for p34 (DM\)
        ld      p3m,#01000001b  ; p34=out(0) p33=in; p35=out p32=in
                                ; p36=Tout p31=Tin; p37=Sout p30=Sin
                                ; p2 push-pull; parity off
```

tended addressing in processors such as the 8088 or 64180.

Special notes: when TALKER is entered, it sends an "M" (for "Monitor") over the serial line. This is the most basic functional test of the hardware.

Also, TALKER puts its own address on the return stack. This means that you can use "go" to start a subroutine, and when the subroutine RETurns, the talker will be re-started. This feature is *very* useful for debugging subroutines.

These six functions are sufficient to do useful work. When I start work with a new CPU, they are all I include in my first talker program.

Breakpoints

Eventually, I'll decide that I really need breakpoint facilities. Fortunately, they are easy to add.

To set a breakpoint, a machine instruction is replaced with a CALL to a breakpoint routine. This breakpoint routine saves the machine state, then enters the debug monitor. The debug monitor must include a command to restore the machine state and resume execution (return from the CALL).

In the talker, all you need to save are the flags and the return address, since the talker uses no other CPU resources. (The six registers used by the talker are reserved for its exclusive use.) Then you just enter TALKER. The "resume" operation depends upon a bit of cleverness, described below.

Four new commands implement the breakpoint facility:

```
HEX ASCII  COMMAND
26   &     set a machine language break-
           point at the MAR address
27   '     copy low byte of MAR to MDR,
           and send to host
28   (     copy high byte of MAR to MDR,
           and send to host
29   )     send the MDR to host
```

The entry point for breakpoints is MBREAK. When MBREAK is CALLed, the flags are copied into MDR, and the return address is popped into MAR. (A similar entry point, IBREAK, pops the flags from the stack. IBREAK can be entered as the result of an interrupt.) MBREAK then sends a "*" to the host to signal that a breakpoint was encountered.

The "go" command has been modified to copy the MDR to the flags register, before jumping to the MAR address. This means that "go" is also the "resume" function. (Note that the stack usage has been carefully arranged to

allow this.)

If any other monitor functions are to be used, the host program must first issue the commands:

```
) ( '
```

to fetch the address and flags, and save them for later resumption of the application program:

```
<high-adrs> - <low-adrs> , <flags> /
```

Why is the "&" command is needed, since the host program could easily use "store memory" commands to build a breakpoint? There are two reasons. First, since I use many versions of this program, the host doesn't know where the MBREAK entry point is located. Second, since I use this monitor for several CPUs, the host doesn't know what opcode to use. Function 26 hex stores the right opcode and the right address for all CPUs.

The host program is, however, responsible for saving the instruction overwritten by the breakpoint. All of the CPUs I use have a 3-byte subroutine call, so the host program just needs to fetch 3 bytes from the breakpoint address and save them. This approach allows any number of breakpoints, just by modifying the host program.

Forth Breakpoints

One of my main uses for this program is to bring up Forth kernels. So, I've added two special kinds of breakpoint for Forth code:

```
HEX ASCII  COMMAND
24   $     set a Forth "thread" break-
           point at the MAR address
25   %     set a Forth "code field"
           breakpoint at the MAR
           address
```

The difference between these two is illustrated by a simple high-level Forth word:

```
: DOUBLE DUP + ;
```

Example 1 shows how this would appear in memory for a direct-threaded Forth. (In an indirect-threaded Forth, the call to DOCOLON would be replaced by just the address of DOCOLON.) Note that DOCOLON is executable machine code, but DUP, +, and ; are Forth words.

A "code field" breakpoint is set by replacing the call to DOCOLON with a call to a breakpoint routine. This causes

```
        ld      p2m,#0ffh       ; p2 all input

        ld      ipr,#00010001b  ; arbitrary irpt priority
        clr     imr             ; all irpts disabled
        ei                      ; must 'ei' to enable IRQ register!
        di                      ; then we can 'di'
        or      irq,#10h        ; set 'tx ready' bit in IRQ for 1st byte

        ld      pre0,#(13*4)+1  ; prescale=13; modulo N timer
        ld      t0,#1           ; for 4800 baud @ 8 MHz clock
        ld      tmr,#00000011b  ; Tout pgmd; Tin clk; t1 off; load & go t0
        ; pre1 and t1 not initialized at this time

        ld      sp1,#0ffh       ; stack at top of RAM
        ld      sph,#0ffh
        srp     #10h            ; wkg regs are 10-1f
        ; ei

        jr      talker

        .endif ; standalone

;=============================================================================
;
;       TALKER MAIN ENTRY POINT
;       MACHINE LANGUAGE BREAKPOINT ENTRY
;
;       This is the main "talker" program.  It calls the basic
;       character processing routine "talk" repeatedly, until a monitor
;       command transfers control to an application program.
;
;       This is also the entry point for machine language breakpoints.
;       A breakpoint consists of a "CALL" to this address.  The call
;       causes the calling address to be pushed on the stack; this
;       routine pushes the flags as well, to allow a common routine to
;       service both CALLs and interrupts.
;
;       Should it be desirable to have an interrupt cause a breakpoint
;       - e.g., a "break" pushbutton wired to an interrupt input -
;       the alternate "ibreak" entry point can be used.
;
;       The breakpoint routine copies the saved flag values into the
;       talker's data register, and the saved return address into the
;       talker's address register.  An immediate "go" function will
;       resume with these saved values (as well as the saved rp).
;
;       Entering the breakpoint routine causes the '*' character to
;       be sent to the host.  Entering the monitor causes 'M' to be sent.
;       Note also that the main entry point its own address onto
;       the stack; this is so that we can "go" to a routine which ends
;       in a RET.  This "normal termination" can be distinguished from
;       a breakpoint by the 'M' character.
;
;       Expects:
;       Returns:
;       Uses:   4 bytes of stack
;
;=============================================================================
        .begin
-pushme: call   -t0
talker: ld      wkg,#'M' ; Talker program entry point
        jr      -pushme     ; some cleverness to push the address 'talker'

mbreak: push    flags   ; Machine language breakpoint entry
        di
ibreak:                 ; Interrupt breakpoint entry
        pop     mdr         ; get saved flags in mdr
        pop     mar         ; get saved adrs in mar (2 bytes)
        pop     mar+1
        ld      wkg,#'*'

        ; All entry points eventually land here
-t0:

-t1:    tm      irq,#10h ; transmitter ready for another character?
```

Example 1

CALL DOCOLON	address of DUP	address of +	address of ;

```
        jr      z,-t1
        and     irq,#0efh ; clear the irq bit
        push    wkg       ; save the 'M' or '*' character during 'txon'
        txon
        pop     sio       ; transmit 'M' or '*' to signal monitor/brkpt
-t2:    tm      irq,#10h ; wait 'til character finished
        jr      z,-t2
        txoff

-t3:    kick              ; at this point tx is off, so kick TXD  v. 1.2CP
        call    talk      ; the talker loop
        jr      -t3
        .end

;=======================================================================
;
;       FORTH LANGUAGE BREAKPOINT ENTRIES
;
;       These are the entry points for Forth language breakpoints.
;
;       The first is the "code field" breakpoint.  This is an address
;       which can be stored in a Forth word's code field, to cause a
;       break whenever that word is executed.  This kind of breakpoint
;       can be set in any Forth word.
;       >>> In the Z8 Direct-Threaded-Code implementation, the
;         parameter field of every Forth word begins with machine code.
;         So, an ordinary machine-code breakpoint can be set in the
;         first 3 bytes of a word.  (All words have at least 3 bytes.) <<<
;
;       The second kind is the "thread" breakpoint.  This is an address
;       which can be patched into a high-level "thread", to cause a
;       break when a certain point is reached in high-level code.  The
;       thread is a series of addresses of Forth words, executed by the
;       inner or "NEXT" interpreter.  So, we provide the address of a
;       dummy Forth word whose execution action is to invoke a machine
;       language breakpoint.  The Forth execution state can be deduced
;       from the registers.
;
;=======================================================================
cbreak: .equ    mbreak  ; the code field breakpoint is simply
                        ; a machine language breakpoint set
                        ; in a Z8 DTC "code field"

; what follows is the parameter and code field of a "headerless"
; Forth word, to invoke the breakpoint routine.  (In DTC, this is
; simply a machine code fragment which does a breakpoint.)

tbreak: call    mbreak  ; the thread breakpoint is simply
                        ; a pointer to this code fragment

;=======================================================================
;
;       TALK - SINGLE CHARACTER PROCESSING ROUTINE
;
;       This routine processes one character received from the host.
;       If no character is received, it returns immediately.
;       It may cause two characters to be transmitted to the host.
;
;       This routine is called repeatedly in a tight loop from the
;       'talker' program, if invoked standalone or by a breakpoint.
;       It may also be called repeatedly from within an application
;       program, as a polled "background" task, to perform monitor
;       functions simultaneously with the application.
;
;       Note - on entry, the stack contains the following:
;                       return adrs in 'talker', lo
;                SP->   return adrs in 'talker', hi
;
;       Expects:
;       Returns:
;       Uses:    flags, regs 0Bh-0Fh.
;
;=======================================================================
        .begin
talk:                           ; check for received character
        tm      irq,#8
        jr      z,~exit

        srp     #regs           ; point r8-r15 to talker's regs

        and     irq,#0f7h       ; clear the irq bit
```

a breakpoint whenever the word DOUBLE is entered, before any of its definition was executed. This kind of breakpoint can be set in any Forth word.

A "thread" breakpoint is set by replacing one of the following addresses with the address of a Forth breakpoint word. This is simply a Forth CODE word which calls the breakpoint routine. If a thread breakpoint were set at "+" above, the breakpoint would be taken after DUP was executed. Thread breakpoints can only be set in high-level Forth words (colon definitions).

Of course, ordinary machine-language breakpoints can be set in CODE words.

The Forth breakpoints are still experimental; I don't yet have the corresponding "resume" functions, and these commands are not included in the host program.

The Host Program

The host program is a set of Forth words which send messages to, and process messages from, the talker program. The user works in the Forth environment, and sees the debugging functions as additional Forth commands.

Figure 2 is a listing of the host program. It is written in MPE PowerForth for the IBM PC, an 83-Standard Forth with the ONLY/ALSO vocabulary extension. Certain functions, namely file access and screen color selection, are specific to this Forth implementation. I hope that the conversion to other Forths is reasonably obvious.

This is an excellent example of building a Forth application by "layering" successively higher levels of abstraction. Starting with the words to perform character I/O, you define primitive talker functions, then more useful operations, until you reach functions such as "alter memory" and "display breakpoint."

Since the code is reasonably straightforward, and much of it is commented, I will present just an overview here.

The Serial I/O

The basic serial I/O functions are ?TX, ?RX, (TX), (RX), TXON, and TXOFF. The latter two deserve comment. Several of my applications have only a bidirectional RS-485 serial port. I've built an RS-232-to-RS-485 converter for my PC, using the RS-232

DTR line to control the direction of the RS-485 transceiver. If you're not running half-duplex, these could be made null functions.

PowerForth uses PC@ and PC! as "fetch byte from port" and "store byte to port", respectively. If your Forth system does not have equivalents, you will have to write CODE words to do this.

Some applications poll the TALK routine infrequently. If characters are sent too rapidly to the target, some of them are lost, and nonsense results. So, PACE allows the transmitter to be slowed in software. This is a crude approach, and open-loop to boot, but it eliminates the need for the target to acknowledge every character.

BAUDTABLE sets up the COM port for a fixed 4800 baud (as required by the Z8 program). This would be easy to alter for variable baud rates.

TERM is a rudimentary terminal program. This is useful to see if the target system is responding at all. Also, some of my application programs (such as Forth kernels) use the serial port for terminal I/O once they are started. I prefer not to exit the host program in order to talk to them.

The terminal program uses a different display color, to show when it's active. The word COLOR uses a PowerForth variable CURR-ATTRIBS, and it will need to be changed for your Forth system. If your Forth doesn't have an equivalent, I suggest:

```
: COLOR  DROP ;
```

Basic Functions

SPILL is used to clean extraneous characters out of the UART. An earlier version of this program did not have this; whenever an extraneous character was received by the host, it remained forever "out of sync" with the target. The program now SPILLs periodically, at times when it is not expecting data from the target.

TXH and RXH send and receive bytes as pairs of hex digits. These are fundamental to almost every talker function.

XADR (external address) uses TXH and the commands 2C and 2D hex, to send an address to the talker's MAR. X@+ and X!+ (external fetch/store with postincrement) implement the commands 2A and 2B hex, respectively. And the words CMEM, EMEM, and REGS are Forth "defined words" which

See Z8 Talker, page 46

```
        ld      r11,sio         ; get character
        and     r11,#3fh        ; mask off all but low 6 bits
        sub     r11,#30h        ; if less than 30 -
        jr      ult,~cmd        ; - it's a command

;
; 30-3Fh: digit entry
;
~digit: swap    r12             ; 30-3f: hex digit, shift into lo nybble
        and     r12,#0f0h
        add     r12,r11         ;   note that r11 has been converted to 00-0f

~done:
~exit:  ret

;
; 24-2Fh: command codes
;       we use a series of 'djnz' tests (simpler & just as economical as
;       a jump table) to select the appropriate function routine
;
~cmd:   add     r11,#(30h-24h+1) ; readjust it upwards for djnz testing

;
; 24H: set a Forth "thread" breakpoint at the given address   (2 bytes)
;       this puts a the address of the "breakpoint" pseudo-word into
;       a Forth thread (a list of addresses of Forth words).  It is the
;       responsibility of the user to ensure that this is a valid thread
;       address, and to save the previous value.
;
~settb: djnz    r11,~setcb
        ld      r11,#^HB(tbreak)  ; store a pointer
        ldc     @rr14,r11
        ld      r11,#^LB(tbreak)  ; to the 'tbreak' Forth word
        incw    rr14
        ldc     @rr14,r11
        decw    rr14
        ret

;
; 25H: set a Forth "code field" breakpoint at the given address  (3 bytes)
;       this changes the code field associated with a given Forth word to
;       point to a machine-language breakpoint routine.  It is the
;       responsibility of the user to ensure that this is a valid code
;       field address, and to save the previous value.
;       >>>For the Super8 DTC Forth, this is the same as function 26H<<<
;
~setcb: djnz    r11,~setmb
        inc     r11             ; fall thru next djnz test

;
; 26H: set a machine language breakpoint at the given address  (3 bytes)
;       this puts a machine-language CALL at the given address.  It is
;       the responsibility of the user to ensure that this is a valid
;       instruction address, and to save the previous value.
;
~setmb: djnz    r11,~getlo
        ld      r11,#0d6h       ; build a 'call' instruction
        ldc     @rr14,r11
        ld      r11,#^HB(mbreak)  ; to the 'mbreak' entry point
        incw    rr14
        ldc     @rr14,r11
        ld      r11,#^LB(mbreak)
        incw    rr14
        ldc     @rr14,r11
        decw    rr14
        decw    rr14
        ret

;
; 27H: copy low address byte to data register, and send to host
;       Note that this destroys the previous data register contents.
;       Use function 2Dh to save that value first, if needed.
;
~getlo: djnz    r11,~gethi
        ld      r12,r15         ; get low adrs byte in data reg
        jr      ~echo1          ; and go send it

;
; 28H: copy high address byte to data register, and send to host
;       Note that this destroys the previous data register contents.
;       Use function 2Dh to save that value first, if needed.
;
```

Local Area Networks

Ethernet

By Wayne Sung

I heard Dr. Bob Metcalfe, the inventor of Ethernet, say once that Ethernet is one of those things that work in practice but not in theory. He was referring to those studies which claimed to show that Ethernet falls apart under even moderate loads. He also said that he probably would not use the term 'collision detect' if he could name the system over, because people get scared by the term.

The Aloha System

To fully appreciate Ethernet, we should first talk about a system called Aloha. You might have guessed that this system originated in Hawaii. This system was designed to provide communications across a large geographic area, where the cost of dedicated lines was simply too high.

In the Aloha system, there is a large central radio transmitter and receiver, with smaller stations in outlying areas. The individual stations could not necessarily receive each other, but all could communicate with the central unit.

If any station had a message to send to the central station, it simply sent it. If two stations sent at exactly the same time, then both messages would be damaged. The central station did nothing about this. Each sending station would see that no reply is forthcoming and try again.

What could conceivably happen is that both stations retry at exactly the same time, thus colliding again. The situation becomes worse if more than two stations tried at the same time. This leads to the collapse that many of the mathematical models of Ethernet predict.

The Metcalfe Improvements

Dr. Metcalfe earned his doctorate by applying improvements to the basic Aloha scheme and thereby greatly decreasing the possibility of collision collapse. These are: carrier sense, collision detect, and random backoff. Put together, these define the CSMA/CD medium access method of Ethernet. The MA refers to multiple access, the primary purpose of LANs.

In the Aloha scheme individual stations typically cannot hear each other. In Ethernet they specifically can. Thus the first rule: carrier sense. Before transmitting, listen to the wire to see if someone else is already transmitting. If so, hold off until they finish.

Even so, because there is propagation delay in any physical medium, collisions can still occur. If two stations fairly far apart on the wire both want to transmit, both may decide that the wire is quiet and send. The actual collision will happen somewhere between the two. Thus rule number two: collision detect. While transmitting, check that all your bits are correctly sent onto the wire.

Originally, collision detect used a bit-by-bit compare of the transmitted and received signals. Unfortunately, some rule changes allowed this scheme to be replaced by a simpler energy-detect scheme. The reasoning went that one station transmitting produces so much energy on the wire, so if more than one transmitted there would be an addition of energy even if the information was garbled.

This usually further reduces to a voltage-detect scheme. Since transceivers used in Ethernet each send a certain amount of current into the wire, this should result in a certain voltage because the wire has a predictable impedance. The rules specify the tolerances that apply.

"It is important to realize that Ethernet is not a guaranteed-delivery system."

If collision is detected, the Ethernet hardware will reattempt the transmission without intervention. What happens if the other guy does the same thing? Here is the most significant part of Dr. Metcalfe's thesis, the truncated binary random backoff. Let's see what that is.

If every station attempted retransmission in exactly the same way, then most likely the collision would reoccur. Thus Ethernet hardware is built so that in the event of a collision, retransmission is attempted at increasingly larger intervals (in fact each interval is double the previous one). The first interval is random, so that multiple collisions can be minimized. After ten unsuccessful attempts, the hardware will give up.

It is important to realize that Ethernet is not a guaranteed-delivery system. This is not an overriding concern, though, because all data transmission systems have some finite possibility of loss. With proper precautions, the overall loss rate can be reduced to arbitrarily small numbers. For the most part this means doing some error checking on the delivered data.

In actual Ethernets, the important thing becomes that all players play by the rules, and the same rules. In many cases, complaints that Ethernet does not sustain high loads is actually due to

Wayne Sung has been working with microprocessor hardware and software for over ten years. His job involves pushing the limits of networking hardware in attempting to gain as much performance as possible. In the last three years he has developed the Gag-a-matic series of testers, which are meant to see if manufacturers meet their specs.

faulty hardware and software, not the theory. I have observed loads over 70% without collision collapse. Some studies claim Ethernet should collapse at less than 30% load.

Since we have moved away from using bridges, we have fewer medium statistics than before. Nonetheless, even on busy Ethernets you would probably find that perhaps one packet in several thousand suffers a collision. On the other hand, deferred packets (the ones where you heard someone already using the network so you waited) are easily ten times that high. These deferred packets do cause some delay, but the delays are measured in units of packet intervals and are seldom noticeable.

Furthermore, we almost never see values other than zero in the retries-exhausted counters. Usually that happens when there is an electrical fault in the network. Even so, when a packet does collide, the retry mechanism is often noticeable, particularly in terminal echo return times.

What If We Break the Rules?

The easiest rule to violate in Ethernet is the length (the physical extent of the network). As a network gets larger, someone on one side of a building may decide it's ok to stretch the length a little. Then someone on the other side of the building decides the same thing. Then someone on the other side of campus...

Ethernet depends heavily on its timing windows to make the collision detect scheme work. This is why Ethernet has *minimum* transmission times. You must stay on the wire long enough to be sure that if a collision occurred anywhere in the network you will have found out about it.

If you stop listening sooner than it takes your transmission to reach the farthest extremes of the network plus the amount of time it takes a collision to come back to you, you may fail to detect the collision. If the network is longer than allowed, then the default times are not long enough, and many people will miss collision detect.

An interesting sidelight to the CSMA/CD scheme is that distance can be traded for speed. For example, by shortening the length of the network to 10% of normal, the signaling rate can be 10 times as high. Thus it is possible to build very fast but much smaller Ethernets, and, in fact, test networks as fast as 150 Mb/s have been tried.

The next most often disobeyed rule is the necessity of continuously checking collision detect during the entire transmission. Some systems assume that if the first part of a transmission succeeded then the rest will also succeed. In fact, for minimum-sized transmissions in maximum-sized networks this assumption is false.

I have seen both hardware and software faults that resulted in not checking collision detect to the end of a transmission. For example, some controllers distinguish between early and late collisions, but the software did not check both conditions.

"Some have decided that the silent interval is for the sake of fairness, so that no one can monopolize the wire. This is not the case."

Each transmission begins with a preamble (unmodulated carrier) of 64 bits. The last 2 bits of the preamble are actually 'start bits'. Then there is a data stream with a minimum of 64 bytes and a maximum of 1518 bytes of information. At the end of each transmission is an enforced silence period of 9.6 us (which would correspond to 96 bit times). The entire transmission is called a packet.

The silent interval is often violated. If you were the last talker, you would normally not attempt to transmit again until the silent interval has expired. What about other stations? Well, some have decided that the silent interval is for the sake of fairness, so that no one can monopolize the wire. This is not the case. The silent interval is to allow all stations to reset their receivers.

Consider station A transmitting a message to station B. If C starts transmitting immediately after A finishes, B may think C's message is a continuation of A's message, since there was not a silent interval to allow B to reset. The only way any station knows a message is over is to receive complete silence for the required period.

This is also the only way a station can know where the CRC bytes are in a packet (they are the last four). If the silent interval is too short or nonexistent, then packets can run together, and hence correctness cannot be determined because the CRC will be in the wrong place. Total received length will not necessarily be excessive, as even 10 packets of 64 bytes run together is still less than the 1518 byte maximum.

One other error condition I have seen involves the use of different sets of rules for the players. Unfortunately there are two sets in common use: Ethernet 2 and IEEE 802.3. In the best case, they do not interfere with each other.

The first twelve bytes of each packet contain the destination and source addresses (each is six bytes or 48 bits long). So far so good. The next two bytes cause trouble sometimes. In Ethernet 2, these two bytes contain a 'type field' which is mainly an identifier of what type of software is in use. 802.3 uses these two bytes to show the length of the information-carrying part of the packet, since not all 64 bytes may be used up.

The two camps informally agreed to a scheme of coexistence. The physical packet cannot be more than 1518 bytes long (actually 1500 information bytes after subtracting the addresses, type, and CRC fields). Thus the 802.3 length field cannot have a value greater than that. It should be safe then to assume that larger values must mean Ethernet 2 is in use.

This scheme works reasonably well, but again all players have to understand the agreement. In most cases, packets have specific destination addresses, so non-addressed receivers who do not understand the scheme do not get bothered. The problem is with broadcasts. Then everyone must look at those two bytes. When the protocols are mismatched, Ethernet 2 devices complain about an unknown protocol, and 802.3 devices complain about the packet being too long.

It takes correctly designed protocols to do a good job over Ethernet, especially when the network gets quite large. It is also possible to run an access scheduler on the network to essentially eliminate the small variability caused by the CSMA/CD mechanism. The token bus system is an example of this: you retain the fast broadcast mechanism of a bus, and use a token-passing method to schedule access so collisions don't happen.

Next time we will look at using broadband cable (CATV) as a distribution medium for LANs and other services. ●

UNIX Connectivity On The Cheap

A Simple Start For CP/M, Z-System, or MS-DOS

By Bruce Morgen

Introducing the power and flexibility of a UNIX or XENIX system into an environment where CP/M, Z-System, or MS-DOS computers currently predominate need not be an prohibitively expensive or technically intimidating. Here are some generalized procedures for connecting popular single-user systems to UNIX that provide a useful level of resource sharing with minimum expenditure and installation effort, creating a connection that allows users of the popular single-user operating systems access to UNIX facilities without giving up the use of familiar programs and procedures for their day-to-day work. With some minimal alterations, these same procedures could be used to link other non-UNIX systems into the UNIX world.

Figure 1

25-pin pin #		9-pin pin #
2	———	3
3	———	2
5	———	8
6	———	6
7	———	5
20	———	4

Specifically, this simple technique makes each single-user system into a UNIX terminal, with the additional advantages of sharing a UNIX spooled printer and optionally transferring files between the UNIX system, which effectively takes the "server" role, and the single-user system. Though not nearly as powerful as a sophisticated mixed LAN environment like SCO XENIX-NET, this level of interaction provides an "entry level" solution that will meet the needs of many applications while providing users with an educational introductory look at UNIX.

The Physical Connection

The first step is to wire the single-user computer to the UNIX server, serial port to serial port. Most microcomputers, regardless of their vintage, come with at least one more-or-less standard RS232 port as standard equipment, and UNIX servers generally have several additional ports through use of a multiplexing multiport board. Although the assumptions in this description apply to the vast majority of hardware, referring to the documentation of the actual equipment is recommended, particularly in the case of multiport cards.

RS232 ports come in two major "flavors," Data Terminal Equipment (DTE) and Data Communications Equipment (DCE). Serial terminals, most CP/M-compatible computers, and almost all PC-style "COM" ports are wired as DTE, modems are generally DCE devices, while printers—and multiport cards—can go either way. Identifying the ports is essential to establishing a working connection. Connecting DTE to DCE requires a "straight-through" cable, connecting DTE to DTE or DCE to DCE requires a cross-over or "null modem" cable. The following discussion refers to standard 25-pin connectors, refer to the conversion chart in *figure 1* if you are dealing with AT-style 9-pin connectors.

Port identification is a relatively simple task if you have a DC voltmeter. A typical analog or digital volt-ohm meter (VOM) or "multimeter" is sufficient. Just set the meter for low DC voltage measurements (in the 0-12 volt range) if necessary, and, with the target PC powered up, connect the "+" (usually read) probe to pin 7, the port's "signal ground." Touch the "-" (usually black) probe to pin 2. A reading of 12 volts or so means the port is DTE, anything close to 0 volts indicates DCE. To confirm that a port is DCE, test pin 3 for a reading in the 12 volt range.

Figure 2

```
Receive Data      3<----------2 Transmit Data
(RD)                            (TD)
Transmit Data     2---------->3 Receive Data
(TD)                            (RD)
Clear to Send     5-+       +-5 Clear to Send
(CTS)               |       |   (CTS)
Data Set Ready    6-+       +-6 Data Set Ready
(DSR)               |       |   (DSR)
Carrier Detect    8-+       +-8 Carrier Detect
(DCD)               |       |  (DCD)
Data Term Ready  20-+       +20 Data Term Ready
(DTR)                           (DTR)
Ground            7<--------->7 Ground
```

Since the vast majority of serial ports on CP/M-style hardware as well as PC-, XT-, and AT-compatibles are DTE, you'll probably be establishing a DTE-to-DTE connection like the one shown in *figure* 2. Note that the CTS, DSR, DCD, and DTR lines are tied to-

A former Associate Editor of Electronic Products, *and columnist for* User's Guide, *Bruce Morgen is currently a free-lance technical writer specializing in marketing and promotional material. He is perhaps best-known for his long-time involvement with ZCPR3, DateStamper, Backgrounder II, XBIOS, and ZSDOS.*

Bruce co-authored Echelon's Z-System User's Guide and "bootable disk" Z-Systems for Kaypros and was on the software development team for the Oneac "ON!" computer. He headed up a Micromint SB180 users group for several years. Bruce has authored several popular Z-System utilities and revised countless others.

A professional musician and songwriter through the 1970s, Bruce remains an avid listener with a keen ear for American and Celtic traditional music. He resides in "ruburban" Bucks County Pennsylvania with his wife Julie and their sons, Ian and Brett. He is co-sysop of Bob Dean's Drexel Hill NorthStar RCP/M (Z-Node #15), 215-623-4040.

gether, a measure not usually necessary for terminal connections, but one often required to access the port as an operating system device, as we will be doing.

You should also take notice of pin 1, the "frame ground." If used at all, this pin should generally be connected at one end of the cable only, as a noise-limiting measure. Don't confuse it with the signal ground line at pin 7, which establishes the common zero-voltage reference point between the two computers and must be connected at both ends. Since we are using a minimal three-wire signal circuit and have not mandated shielding, limiting the cable length to 50 feet as specified in the RS232 standard is strongly recommended.

Establishing a UNIX Terminal

With the physical connection completed, the next step is the selection of the terminal emulation software. Various public domain, commercial, and "shareware" serial communications packages—e.g. QTERM and MEX+ for CP/M or Z-System, ProComm, Crosstalk, and MEX-PC for MS-DOS—are well-equipped for this role. The choice is yours. Set the terminal emulator's serial port characteristics—baud rate, data word length, stop bits, and parity—to match those in effect at the connected UNIX port. Consult the UNIX server's system administrator if you don't know these parameters.

We then have to introduce the our "terminal" to UNIX. Procedures for doing this are in the UNIX system documentation. As an example, for SCO XENIX Release 2.2 consult Chapter 7 of the Operations Guide, for SCO's Release 2.3 the required information is in Chapter 14 of the System Administrator's Guide. With the physical connection complete and the appropriate software set up on both systems, the user should be able to log in as a UNIX user. When that is accomplished, we can go on to implementing the single-user-to-UNIX printing process.

Setting Up for Printing

Once users are able to perform the UNIX log-in sequence when they start up their systems each day, they can check for UNIX mail, read the system news, check their personal calendars, or take advantage of whatever other UNIX facilities are available on the server. It important to emphasize to *not* log off UNIX when exiting the terminal emulator, because an active shell prompt is required in order to print via UNIX. Logging off should be done at the end of the work day or at the request of the system administrator.

I'll actually present two printing sequences, the first of which is for WordStar files, the second for straight ASCII printouts. If you want use another application program, your chance of success are good with the appropriate changes to the examples, but you are on your own.

Setting The Stage

Each CP/M or Z-System computer may need one or more device selection and initialization commands issued. Depending on the individual system, these can be done with the CP/M STAT utility or (if an appropriate ZCPR3 I/O Package is loaded) the Z-System DEV command. Under Z-System, these commands can be added to the startup alias with an alias editor/generator like SALIAS. A typical STAT command might be:

```
STAT LST:=TTY:
```

Each MS-DOS system will require one or more MODE commands added to DOS's AUTOEXEC.BAT start-up script with EDLIN or another non-document DOS text editor. If you are using the COM1 port, the first of these lines should be:

```
mode lpt1:=com1:
```

We cannot depend on the single user system's default serial port settings matching those on the UNIX server, so our next addition to the Z-System startup alias or DOS's AUTOEXEC.BAT should specify them exactly, just as we've already done for the terminal emulator. CP/M and Z-System have no standard syntax for doing this and appropriate utilities are usually hardware-specific—Ampro uses SET.COM, Kaypro uses BAUD.COM, et cetera. On an MS-DOS PC, if the UNIX port is set for 9600 baud, no parity, 8 data bits, and 1 stop bit, the DOS command line should be:

```
mode com1:96,n,8,1
```

Strings For Unix

Now we need to create two text files containing UNIX commands that will start and conclude the printing process on the server.

CP/M and Z-System users can use the console input function of PIP to produce the first of these files, LPOPEN:

```
A0:BASE>pip lpopen=con:<CR>
cat | lp<CR>
<CTL-Z>

A0:BASE>
```

Under MS-DOS you can use the console input function of the COPY command as follows to produce LPOPEN:

```
C>copy con: c:\lpopen<CR>

cat | lp<CR>
<CTL-Z>

C>
```

Now make the second file, LPCLOSE. Under CP/M or Z-System, do:

```
A0:BASE>pip lpclose=con:<CR>

<CR>
<CTL-D>
<CTL-Z>
```

Under MS-DOS do:

```
C>copy con: c:\lpclose<CR>

<CR>
<CTL-D>
<CTL-Z>
```

Finally, we must create the Z-System alias LP, or the DOS batch file LP.BAT, which will start the printing process. The syntaxes given assume WordStar Version 4.0 or later, as well as the availability of PIP/COPY. Vanilla CP/M die-hards will have to concoct a SUBMIT file,restricting their operations to a single user area—just eliminate any directory change lines from the Z-System example below and assign drive letters to file specifications as required.

See Connectivity, page 30

The PC Hard Disk Partition Table

by Rick Rodman

From the early days of hard disks on the IBM PC-XT, PC hard disks have had what is called the "partition table". Yet, to my knowledge, the structure of this table has never been documented, nor have programs been presented to access or modify this table.

DOS, whether PC-DOS or MS-DOS, comes with a ridiculously cumbersome and stupid program called FDISK. It appears to have been designed intentionally to frustrate and frighten people—maybe so they'll take the machine back to the dealer. Programs intended to "protect the unsophisticated user against himself", in my mind, come from an elitist mindset I find very offensive.

Minix 1.5 includes a much better FDISK program, and the discussion in the manual, and the prompts issued by that program, are the source of much of the insight presented here. However, much of what is here was determined through study and experimentation: fooling around with a hard drive for hours, and studying FDISK internals.

So, once again *TCJ* breaks the information cartel and dares to bring you the true facts about the partition table. Save this issue, because who knows when—or if—you'll ever see this information in print again.

Why is there a partition table?

The partition table allows different sizes of hard disks to be used in the system, with automatic configuration. Its function is somewhat similar to the Boot Parameter Block (see "Mysteries of PC Floppy Disks Revealed", issue #44) in this regard. However, the partition table is really a layer below the BPB. One of its functions is to divide a hard disk into multiple "logical drives".

Another function of the partition table is to allow different parts of the hard disk to be used under different operating systems. For example, the first hard disk of the system I am using to write this has a single DOS partition and two Minix partitions.

How the partition table is used under DOS

A DOS-only hard disk generally has two partitions. The first is called the "Primary DOS" partition. This must be the first partition on a disk if it is to boot DOS. The Primary DOS partition can contain only one logical drive. The second is called the "Secondary DOS" partition. This partition is divided up (by a means outside the scope of this article) into multiple logical drives.

Where the partition table is located

The partition table is the first sector of a hard disk. It is on cylinder (track) 0, head 0, sector 1, that is, the first sector. It seems to be a PC tradition to number disk sectors starting at 1, although heads and cylinders (tracks) are numbered starting at zero. Don't ask why, it's just tradition. Another little-known (and little-documented) fact of the AT BIOS is that the hard disk drive identifiers are numbered starting at 128.

Structure of the partition table

The partition table is located at the very end of that first sector. The first part of that sector usually includes some boot code. The table begins at offset 1BE hex in the sector, and contains four entries, allowing up to four partitions on a single hard drive.

Each record of the partition table has the following general structure:

Offset	Bytes	Contents
0	1	Boot indicator
1	1	Starting head
2	1	Starting sector
3	1	Starting cylinder
4	1	Operating System ID
5	1	Ending head
6	1	Ending sector
7	1	Ending cylinder
8	4	Relative sector
12	4	Total sectors in partition
	16	bytes in partition record

The four partition records are at offset 1BE hex, 1CE, 1DE, and 1EE hex. At offset 1FE hex (the last two bytes of the sector), a two-byte indicator value 55 AA hex is stored to indicate the presence of the partition table.

The structure is unfortunately not as simple as it looks. Larger hard disks with more than 256 cylinders made necessary some modifications to this scheme, and these modifications were made in the screwy way that assembly-language programmers are fond of. The following discussions of each field will note these oddities.

Boot indicator: This value is supposed to be set to 80 hex to indicate a partition as being bootable. However, it doesn't work. The BIOS will attempt to boot only the first partition, and only if it contains a DOS-style BPB.

Start head: This is the first head used in a partition. Basically, each partition is a "chunk" of the hard disk which goes across all heads of the drive for a range of cylinders. There is no other way of knowing how many cylinders the drive has, so the start head used for all partitions must be zero, and the end head must always be the number of heads minus one.

Some operating systems may support a partition not ending on the last head, or not ending on the last sector, last head of the last cylinder of the partition. But exercise caution; I don't think DOS 3.3 is that smart.

Start sector: The low-order six bits of this byte store the

first sector of a partition. That is to say, the sector number of the first sector used on the start head, on the start cylinder, of the partition. Usually this value is 1, but the first partition must start at 2 to leave room for the partition table itself.

The upper two bits of this byte are an extension of the start cylinder field, to allow it to specify a cylinder of up to 1023.

*Start cylinder:*This field contains the low-order eight bits of the starting cylinder of a partition. The high-order two bits are stored in the start sector field.

Operating System ID: This field identifies the type of the partition. The values are not standardized by any known body and appear to have been arbitrarily chosen by people as they put OSs out there. The following are a few values known to have been seen in this field (hexadecimal):

```
01 Primary DOS with 12-bit FAT entries
02 Xenix?
03 Xenix?
04 Primary DOS with 16-bit FAT entries
05 Extended DOS with 12-bit FAT entries
06 Extended DOS with 16-bit FAT entries
07 HPFS (OS/2 1.2 and later)
08 AIX
51 Novell?
52 CP/M-86?
63 386/IX?
64 Novell?
75 PC-IX
81 Minix
DB CP/M-86?
```

If you have any additions or corrections to this table, please share them with your fellow readers. (Note that some of the values given by Minix 1.5 are listed here; on the other hand, some of the values given by Minix 1.5 are clearly wrong, too.)

Ending head: This value is the ending head of the cylinder and appears to be required to be the last head of the drive.

End sector: This is the sector number on the last head of the last cylinder of the partition. Note again that only six bits are used. The upper two bits belong to the next field, End Cylinder.

End cylinder: This is the last cylinder of the partition. The eight least-significant bits are stored in this byte; the upper two bits are stored in the End Sector byte.

Relative sector: This is a long, 4-byte unsigned integer which represents the count in sectors from the start of the drive to the beginning of the partition. The first partition will always have

See Partition, page 55

```
/* READPART.C - Read partition table
        901219  rr      from readabs.c
        901224  rr      get working

        For Datalight Optimum-C
*/

#include "dos.h"
#include "stdio.h"

#ifndef I8086L
** error must be compiled large model
#endif

unsigned char buffer[ 1024 ];

static int read_partition_table( void );
static void check_drive_type( void );
static void show_partition( int, unsigned char * );

static int      drive_id = 128;

/* - main program - */

main( int argc, char *argv[] ) {
        int             n;
        char            tbuf[ 80 ], *p;

        if( argc > 1 ) {
                drive_id = atoi( argv[ 1 ] );
        } else  drive_id = 128;

        memset( buffer, 0, 1024 );

        check_drive_type();

        n = read_partition_table();
        printf( "result of read is %d\n", n );

        if( n == 0 ) {
#if 0
                /* print buffer as 21-1/3 lines of 24 */

                for( n = 0; n < 512; ++n ) {
                        if( n % 24 == 0 ) printf( "\r\n%04x :", n );
                        printf( " %02x", buffer[ n ] );
                }
#endif
                show_partition( 0, &buffer[ 0x1BE ] );
                show_partition( 1, &buffer[ 0x1CE ] );
                show_partition( 2, &buffer[ 0x1DE ] );
                show_partition( 3, &buffer[ 0x1EE ] );
        }
}

/* read partition table */

static int read_partition_table() {
        int             retry, n;
        union REGS      regs;
        struct SREGS    sregs;

        for( retry = 0; retry < 3; ++retry ) {

                sregs.es = sregs.ds = getDS();

                /* Drive = 0 = A:, 1 = B:, 128 = C:, 129 = D: */

                regs.h.dl = drive_id;
                regs.h.dh = 0;          /* head */
                regs.h.ch = 0;          /* cylinder */
                regs.h.cl = 1;          /* sector number (0?) */
                regs.h.ah = 0x02;       /* read sector(s) */
                regs.h.al = 1;          /* 1 sector */
                regs.x.bx = ( unsigned int ) &buffer[ 0 ];
                                        /* buffer */
                int86x( 0x13, &regs, &regs, &sregs );
                                        /* invoke BIOS */

                n = regs.h.ah;          /* get errors */
```

A Short Introduction to Forth

By Frank Sergeant

The Forth System

Forth is easy to grasp once a few things are explained. First, keep in mind three of the parts of a Forth system: the stack, the dictionary, and the input stream. Numbers go on the stack. Named, executable routines go in the dictionary. Characters you type from the keyboard (forget the disk for now) go into the input stream.

The Interpreter

Forth stuffs keystrokes into the input stream until you press return, then it interprets that input stream one word at a time. It takes each word and tries to look it up in the dictionary. If found in the dictionary, the word is executed. Otherwise, Forth tries to convert it to a valid number. If it can do so, it pushes the number to the stack. Otherwise, it reports an error. When the input stream is empty, Forth starts over collecting keystrokes and then interpreting them.

The Stack

Numbers are kept on the stack. Sometimes this is called the data stack or the parameter stack. It is the central clearing house for numbers. Every word that needs numeric input knows where to find it: on the stack. Every word that supplies numeric output knows where to deliver it: to the stack.

Words

The basic unit of Forth is the *word*. A word is any group of characters separated by white space (one or more spaces or carriage returns). Thus, there are seven words on the following line:

```
@       !   12345   DUP   SWAP   :   ZZQQDI
```

Forth words are *active*. They don't *mean* something; they *do* something. That is, they are not symbols the compiler or interpreter looks for to determine what to do. Instead, each Forth word just plain *does* it. For example, the Forth word + (pronounced "plus") is not a symbol indicating addition should be done; it *does* the addition. + removes the top two numbers from the stack, does the addition, then puts the sum onto the stack.

This active nature of Forth words is a key concept. It gives Forth much of its modularity and power. Each little piece in a Forth program is an active, independent unit.

Even Forth's comment indicator, the left parenthesis, is active. It doesn't signal a comment, it actively gobbles up the input stream until it finds a right parenthesis. Thus, comments start with a left parenthesis *followed by at least one space.*

Numbers

How, you might ask, do those two numbers get onto the stack in the first place? The answer, from the viewpoint of + is that *it doesn't care*! This is important because it eliminates dependencies and side effects between words. But, even if + doesn't care, you might. There are several ways to get numbers on the stack. Earlier words can put numbers on the stack, or, you can just type them. Remember the 3-way process of the interpreter? When it can't find the word in the dictionary, it tries to convert it to a number and put it on the stack. So, typing **3 4** would put those two numbers on the stack.

Isn't "reverse Polish" difficult? No, no, this "postfix" is very simple. It just means the operator (the active Forth word, for example) follows its operands. This is how to add 4 and 7:

"Isn't 'reverse Polish' difficult? No, it just means the operator follows its operands."

```
4 7 +
```

The + would remove the 4 and 7 from the stack and return an 11 to the stack.

Many Forth words take two operands. In some cases like + and * and AND, their order makes no difference. In others, such as - and /, the order of operands is very important. It is easy to get the order right by remembering that it is the *same* as you are used to. The only difference is the location of the operator; the order of the operands remains the same. For example, the algebraic 15 / 3 would be written in Forth as **15 3 /** meaning in both cases to divide 15 by 3, giving 5 as the quotient. This is called "postfix" because the operator comes after the operands, rather than "infix" where the operator goes *in* between the operands. In a similar manner, **7 3 -** is the Forth way of subtracting 3 from 7.

Programs

In Forth, rather than just writing a program, you extend your system by building tools to make writing your program easier. Finally, your program can be written in one or two lines. More powerful words are built up from simpler words. These new words, in turn, are used in the definition of other words. As you progress you customize the language to the specific task at hand. Each word is simple and can be tested easily from the keyboard, making testing a dream instead of a nightmare.

Forth systems come with many words already defined in

the dictionary. These include words that allow you to define your own additional words. The word : (pronounced "colon") puts a new word into the dictionary. For example, suppose you want to define a word named **3*** to multiply a number by 3. Here is what to type in:

```
: 3*    3  *  ;
```

The : creates a dictionary entry for the new word **3*** and then compiles the following words up to the semicolon. The word following colon, i.e. **3*** , becomes the name of the new word. The rest of the words, namely **3** and ***** are laid down in the dictionary as the actions that the word **3*** will perform.

Now that **3*** has been added to the dictionary, you can type **17 3***. The **3*** will put a 3 on the stack, then consume the 17 and the 3, and leave 51 on the stack. This new word is a *full citizen*. It is indistinguishable from words in the dictionary that came built in. It behaves just like the rest of the Forth words: it takes its arguments from the data stack and puts its results back onto the data stack. This feels different from languages where the functions and procedures and subroutines are second class citizens. There are no "reserved words" in Forth.

Stack Comments

When a new word is defined it is helpful to show a picture of its effect on the stack. This is called a stack comment. It is usually placed inside a comment within the new word's definition. The left side shows the "before" condition of the stack, i.e. what the word expects on the stack. The right side shows the "after" condition, i.e. what the new word leaves on the stack. There must be some separator so you can tell where the "before" ends and the "after" begins. I use a single hyphen as the separator, some people use a double hyphen. Don't confuse this with a minus sign or Forth's subtract word. Since the stack comment *is* a comment, you can write it any way you wish as long as you follow the left parenthesis with at least one space. Here is the definition of **3*** rewritten to include a stack effects comment:

```
: 3*  ( n - 3*n)          3  *  ;
```

When you know the stack action of a word, you know everything (within reason). In the above example, n represents any number and 3*n represents three times that number. Sometimes throughout the definition you'll find additional comments showing what is on the stack at that point. If the stack picture gets too complex, perhaps the programmer should break the definition into several smaller, simpler definitions.

Building Blocks

A Forth word can be thought of as both a program and a subroutine. You can execute it from the keyboard or you can use it as a building block within another word. Contrast this with the difficulty of testing a single procedure or subroutine in most languages.

There is no distinction between words built in to the system, words you add, programs, subroutines, et cetera. In Forth they are all just *words*, and can be executed from the keyboard or as part of the definition of another word.

This means several important things. It means a Forth system can have a number of "programs" available in memory for immediate use. Often Forth can serve as a self-contained environment, allowing immediate access to the editor, assembler, debugging tools, plus all the routines you've defined.

This ability to exhaustively test each little piece from the keyboard adds robustness (and flavor). That is, you build programs on a solid foundation of tested and certified elements. Proper Forth programming style is to build your program as small, easily tested pieces. Don't write huge procedures, as we often see in Pascal, for example. Ideally, each word's definition will take a single line, as in the **3*** example.

These little pieces are reusable. As you program you gradually customize the language to suit exactly what you want to do.

Funny Names

Frequently used Forth words are usually short. This makes them easy to read and type. But, until you know what they mean they can stand in the way of your understanding Forth source code. You need a glossary you can refer to. **C@ C! @** and **!** are four examples. They are pronounced "C-fetch," "C-store," "fetch," and "store." The first takes an address from the stack and returns the byte value located at that address. It is similar to PEEK in BASIC. The second takes a byte value and an address and stores the value at the address, rather like POKE in BASIC. The "C" in **C@** and **C!** stands for "character" and indicates byte size values. **@** and **!** work the same way, but with 16-bits values.

The **.** (pronounced "dot") is a word that takes a number off the stack and displays it. Thus, the dot is associated with printing or displaying. Often it is used in the names of words that have something to do with displaying or printing. For example, the word **."** ("dot-quote") prints a string up to the ending quote mark. It is similar to PRINT" in BASIC.

IF THEN WHAT?

Some people seem to have trouble with Forth's **IF** example:

```
: TRUE?  ( flag -) IF ." YES" ELSE ." NO" THEN  ;
```

IF is an active word that takes a number off the stack. If the number is "true" (i.e. non-zero), then the part between **IF** and **ELSE** is executed, otherwise the part between **ELSE** and **THEN** is executed. **THEN** marks the end of the entire structure, where both paths converge. If it gives you trouble, read **THEN** as "end-if".

Loops

Loop structures of one sort or another are usually pretty obvious, if you remember that **WHILE** and **UNTIL** each take a truth value off the stack to decide whether to branch or not.

There are two main counted loop structures in Forth. These days I only use **FOR ... NEXT**. For example,

```
: STAR ( -)     ." *"  EMIT  ;
: STARS ( n -)  FOR  STAR  NEXT  ;
7 STARS  *******
```

It works much like you'd expect it to. In some Forths **7 STARS** would print 7 stars and in others it would print 8 stars. Hopefully the ones that print 8 will change their ways.

The other is **DO ... LOOP** as in

See Intro to Forth, page 32

REAL COMPUTING

The 32CG160, Swordfish, and the DOS command processor

By Richard Rodman

AMD has been using its 29000 RISC processor to try to capture the high-end embedded systems market from Motorola and National. They have targeted both the VME-bus market and the laser printer market. As an example, Hewlett-Packard's new IIISi printer uses the 29K. In the meantime, Motorola is targeting Sun and SPARC with its 88000 (while selling 68040s to NeXT and Apple), and Intel is still wondering what to do with its 80860.

National, meanwhile, is trying to capture the mid-field, where high integration and low cost are expected to move great quantities of parts against competition from the 68332 and the 80386SX.

The 32CG160

The 32CG160 is a new part oriented towards embedded computing. It takes the 32CG16 CPU, with its additional graphics logic, and, on the same die, adds a Bit-BLT processing unit, a two-channel DMA controller, a simplified ICU including three 16-bit programmable timers, and an on-chip clock generator. It can use an FPU but not an MMU. It is housed in an 84-pin PLCC. It also includes a high-speed multiplier similar to that on the 32GX320.

Based on the projected costs of this chip, it should be very popular in high-volume applications. It should also be a popular choice for low-priced X terminals.

X terminals, by the way, are the sleeper market of the nineties. While the pundits of the press continue to say that there is little use for them, they are quietly shipping in surprisingly large volumes. Wouldn't it be funny if the industry moved back to separate computers and terminals again? In fact, wouldn't it be something if PC operating systems could take care of the device dependencies and the application software wouldn't have to write directly to the hardware? Wow, it could be like... CP/M again!

At any rate, with all of the apparent convergence in Motif, Xt, PM, Windows, and Open Look, I think there could be a really neat board with a CPU, Xlib, and VGA emulation on it, which plugs into a PC and runs ho-hum VGA or screaming X. If you use TIGA it'll cost $3000; if you use the 'CG160, it could cost $300. Somebody do it. Please!

But we interrupt these niceties for some real news.

Swordfish

CISC has blown away RISC once again. National has finally released the production version of what was to be the 32764. It has two complete CISC integer processors on it, which interleave the processing of instructions over 64-bit busses to achieve 100 MIPS.

Each of the two CPUs has its own four-stage pipeline. An on-chip FPU allows integer and floating point operations to be done in parallel. Yet all of this complexity is managed without requiring the programmer to jump through hoops, as with a RISC. National compared the performance of the Swordfish (actually called the 32SF640) with DSPs. Although its floating-point performance of 20 MFLOPS isn't that outstanding compared to other DSPs, remember that this chip has the ease of programming of a general-purpose CPU, and 100 MIPS integer performance.

There appear to be some additional enhancements over the other NS32 processors. The chip contains additional registers which can be used for interrupt handlers, eliminating the need to save and restore them (shades of the Z-80!). The chip also includes a single 16-bit counter timer, a two-channel DMA controller, and a simplified ICU.

The chip is packaged in a (get this) 223-pin PGA. Prices are not cheap for this processor. There is an evaluation board for $10,000 from National, who is also offering a specially optimized C compiler for the VAX and for the Sun.

Personally, I've always considered RISCs and DSPs as temporary aberrations along the road to faster general-purpose, easy-to-program CPUs. We'll be seeing more GaAs (gallium arsenide) hardware soon, too. But someday soon, things will be good. Good enough to where, for example, if you want to track objects in real-time video, you'll just sit down at your general-purpose PC and write a program in your favorite language. The Swordfish is a big step in the right direction.

The DOS command processor

A useful feature of the DOS command processor is that environment variables can be used as a form of "logical name", but only in batch files. For example, if the environment variable COMPILER has been set to a directory where the compiler resides, a batch file could contain a line:

```
%COMPILER%\cc %1.c
```

Oddly, however, this won't work on the normal command line. Remember never to use spaces in SET commands under MS-DOS. Everything starting

Rick Rodman works and plays with computers because he sees that they are the world's greatest machine, appliance, canvas and plaything. He has programmed micros, minis and mainframes and loved them all. In his basement full of aluminum boxes, wire-wrap boards, cables running here and there, and a few recognizable computers, he is somewhere between Leonardo da Vinci and Dr. Frankenstein. Rick can be reached via Usenet at uunet!virtech!rickr or via 1200 bps modem at 703-330-9049.

with the first nonblank character after the word SET is stored in the environment, verbatim. So, if you would happen to enter spaces in there, those spaces become significant.

Another oddity is the fact that the FOR command is different between batch files and the command line; in a batch file, two percent signs must be used before the variable, but on the command line, only one may be used.

In working on the command processor for Bare Metal, these idiosyncrasies have all been corrected. (Why emulate COMMAND.COM instead of, for example, the Bourne shell? Because it's easier to do, and easier to use.)

Of course, these aren't the most glaring deficiencies of DOS' command processor. The most glaring deficiency, the inability to nest batch files, was corrected in version 3.3 with the CALL statement, although the reason for requiring the additional word is unclear. What's really amazing is that, after ten years, Microsoft has still not addressed the following severe deficiencies:

1. No BEGIN/END for multi-line FOR or IF blocks.
2. No lexicals for string parsing, e.g. to separate drives and directories.
3. No real CALL/RETURN for subroutines within the same batch file.
4. ECHO OFF echos to the screen.
5. I could go on, but why bother.

It seems as though Microsoft's priorities lie elsewhere than in developing the command processor. IBM, on the other hand, is supposedly bundling REXX with OS/2 2.0, which is a real command processor. This is a good move. Anyone who has done any programming for GUIs knows that it is extremely tedious to program under a GUI—you simply must have a command processor.

A common oddity of both DOS and OS/2 is that, although directories are files, they cannot be accessed as such. Instead, special search first/search next calls must be used. There's no logical reason for this that I can see, so it must be a quirk of the internal implementation. (Remember under DOS 2.x, you couldn't access read-only files unless you used FCBs?) This has caused much difficulty in Bare Metal's remote manager logic and is delaying release of version 0.7.

Ideally, of course, you want the application programs to access the directory via the operating system and not have to worry about the structure of directory entries, especially in an operating system supporting multiple different file system structures, as does Bare Metal.

OS/2 1.3

On my OS/2 system, I recently upgraded from version 1.2 to version 1.3. The improvement in speed from this upgrade is really startling. Screen operations, like scrolling of text in windows, repainting, and other graphics operations, are several times faster. As I mentioned before, any DOS program I've tried, including Windows 3.0, will run in the DOS Compatibility Box. But by producing OS/2 versions of my NS32 tools, the compiler, linker and assembler, I can now run these in one session while editing in another, and watch the output scroll in a window. You can't do that under Windows, buddy.

I've recently heard that programmers are actually developing Windows programs under OS/2, using the Software Migration Kit (now called WLO), then porting them back to Windows. They do this to gain the multitasking and protected memory modes of OS/2. I could rant and rave about the bizarre modes and memory models of Windows...but let's just say Windows is like sausage: it sizzles and makes your mouth water, but don't ask about its internals.

Anyway, anyone thinking about putting Windows 3.0 on a system ought to give some thought to OS/2 1.3.

Next time

Minix may be getting its own GUI. We'll check that out, plus other developments on the Minix, Mach and Amoeba fronts. In the meantime, please share your insights with your fellow *TCJ* readers.●

Connectivity, from page 24

Here's the Z-System alias text for LP. It assumes LPOPEN and LPCLOSE are in the named directory BASE:

```
BASE:
PIP LST:=LPOPEN
$D$U:
WS $1 PX
BASE:
PIP LST:=LPCLOSE
$D$U:
```

Under MS-DOS, we can once again enlist COPY, this time to create the batch file LP.BAT:

```
C>copy con: c:\lp.bat<CR>

copy c:\lpopen coml:<CR>
ws %1 px<CR>
copy c:\lpclose coml:<CR>
<CTL-Z>
```

With these files in place, reboot the single-user system (vanilla CP/M users may also need to enter some commands manually) and you'll find you can print WordStar files via the UNIX server with your new "LP" command. To print the file MYFILE.WS, enter this:

```
C>lp myfile.ws<CR>
```

The first PIP/COPY command of LP executes a UNIX command that will take input from the connected port and route it to the XENIX print spooler, "lp." Then WordStar is executed with MYFILE.WS as its input file and "px" as trailing parameters that tell WordStar to print the file and exit. Because of the redirection implemented by the commands in the startup alias or AUTOEXEC.BAT, the printer output will go to the serial port and become the input to the previously set up UNIX command. The last line sends carriage return and a <CTRL-D> to the UNIX command line to indicate that we've completed our input to the print spooler.

The LP command can be run from WordStar's opening menu via the "R" option if available memory allows. The WordStar printer driver must match the UNIX printer for correct printed output, and, depending on the printer, certain WordStar print controls may have be avoided, as determined by experimentation.

A variant of LP for unformatted printing of plain ASCII text is next on the agenda. We'll call it LPA, and its attendant text files LPOPENA and LPCLOSEA. Here's how to create LPOPENA (please note the "grave accent" marks on the first line—this character is usually found sharing a key with the tilde ("~") character):

```
A0:BASE>pip lpopena=con:
      (OR)
C>copy con: c:\lpopena<CR>

oldstty=`stty -g`;stty igncr -icrnl;cat|lp<CR><CTL-Z>
```

To make LPCLOSEA, do:

```
A0:BASE>pip lpclosea=con:
      (OR)
C>copy con: c:\lpclosea<CR>

<CTL-J><CTL-D>stty $oldstty<CTL-J><CTL-Z>
```

For Z-System, make the alias LPA:

```
BASE:
PIP LST:=LPOPEN
$D$U:
PIP LST:=$1
BASE:
PIP LST:=LPCLOSE
$D$U:
```

Finally, here's how to create LPA.BAT using COPY under MS-DOS:

```
C>copy con: c:\lpa.bat<CR>

copy c:\lpopena coml:<CR>
copy %1 coml:<CR>
copy c:\lpclosea coml:<CR>
<CTL-Z>
```

The reason for the various "stty" commands in LPOPENA and LPCLOSEA is that UNIX normally converts incoming carriage returns to newlines (otherwise known as linefeeds), which, when added to the newlines already in CP/M- and DOS-style ASCII files, would cause the text to be printed out double-spaced.

The first line of LPOPENA saves the initial characteristics of the UNIX port, which are correct for terminal operations, the second tells UNIX to ignore carriage returns and to not map them to newlines, thus effectively filtering the input from DOS to UNIX text file conventions. The last line of LPCLOSEA restores the original, normal UNIX terminal port characteristics after the input has been spooled for printing.

The examples given for Z-System refer to the named directory BASE. Z-System savants will recognize this as arbitrary—feel free to alter as required to your directory layout and/or preferences.

Similarly, the example MS-DOS files were all placed in the DOS root directory on the system's primary hard drive. If you understand how MS-DOS pathnames work, then you can place these files where you please as long as you modify the references in the associated batch files accordingly.

Possible Problems

Most glitches will be the result of users attempting to run a print job when they were not logged into the UNIX server. This is usually benign, because UNIX will reject each line of printer output as an incorrect login attempt. At worst, some versions of UNIX will ban the user from the server pending system administrator intervention. A similar, but potentially much more dangerous, situation could occur if the user exited the terminal emulator while a UNIX application was still active. In that case, the harm done would depend on how the application responds to the input.

Most problems with strangely formatted printer output can be corrected with "stty" commands along the line of those used in the LPOPENA and LPCLOSEA examples. The UNIX documentation contains pages of information on stty, a powerful command that should be used with the utmost caution (if at all) by inexperienced UNIX users.

Adding File Transfer

File transfer is another facility supported by a simple serial UNIX connection. Only two things are required: a UNIX-compatible version of the public domain XMODEM program on the server, and XMODEM capability on the DOS side, a feature built into almost all CP/M and MS-DOS modem communications software. Depending on the vintage of the UNIX-compatible XMODEM program, a number of variants of the error-correcting XMODEM protocol are supported, with the original Christensen version (128-byte packets and simple checksum error testing) being the lowest common denominator supported by all XMODEM implementations.

Consult your XENIX XMODEM program's built-in help screen and your communications software's documentation for more information on file transfer via XMODEM. Many communications packages support batch-style command scripts that can simplify and automate file transfer operations, check your package's documentation for further details.●

Intro to Forth, from page 28

```
: STAR ( -)  ." *"  EMIT  ;
: STARS ( n -)   0 DO  STAR  LOOP  ;
7 STARS *******
```

The difference is that **FOR ... NEXT** takes a single number and **DO ... LOOP** takes a limit and a starting number.

Stack Manipulation

Sometimes we need to re-arrange the numbers on the stack. Forth has a collection of words to manipulate the stack, such as the following: **DUP** copies the top item, **DROP** throws away the top item, **SWAP** reverses the positions of the top two items, **ROT** moves the third item to the top (thus **ROT ROT ROT** would leave the stack unchanged). Here's another way to define **3*** that uses **DUP**

```
: 3*  ( n - 3*n)  DUP DUP ( n n n)  + ( n  2*n) +  ( 3*n) ;
```

A BASIC Rosetta Stone

Here are some sample Forth phrases with their equivalents in BASIC:

```
." HELLO "              PRINT "HELLO";

3 .                     PRINT 3;

TEST                    GOSUB TEST

3 7 + .                 PRINT 3+7

3 FOR ." H"  NEXT       FOR I=3
                              PRINT"H";
                        NEXT I

( This is a comment)    REM This is a comment

KEY EMIT                100 LET A$=INKEY$:IF A$=0 THEN 100
                        120 PRINT A$;

KEY .                   100 LET A$=INKEY$:IF A$=0 THEN 100
                        120 PRINT ASC(A$);

VALID?
IF RTNA ELSE RTNB THEN  IF VALID THEN GOSUB RTNA
                              ELSE GOSUB RTNB

3721 C@ .               PRINT PEEK(3721);

65 3721 C!              POKE 3721,65
```

Speed and Assembly Language

Forth is the fastest language. Bear with my little joke. In high-school, a friend was a sports car enthusiast. He said no matter how fast your car was, a cop car was always faster—because it had this special speed device—called a "Motorola." By that he meant the two-way radio whereby the cop could radio ahead. How does that make Forth the fastest language? Well, it really only makes it equal to the fastest. In Forth it is very easy to drop down to assembly language any time greater speed is needed. It is almost never the case that every part of a program needs to run flat out. Usually there are one or two key routines that make or break the program, speed-wise. One of the beauties of Forth is you can code the whole program, without using assembly language, and test it. When it is working, you *replace* those few key routines with assembly language if you really need greater speed. The program looks and works the same. No special calling conventions are needed.

So, suppose it turns out that **3*** is the bottle neck in your program. If only it were faster all would be well. You could re-write it in 8088 assembly language as follows:

```
CODE  3*   ( n - 3*n)
  BX AX MOV,   BX BX ADD,   AX BX ADD, NXT,
 a

END-CODE
```

Don't worry too much about the assembly language. I just want to get across the general idea. **CODE** and **END-CODE** correspond to the beginning colon and ending semi-colon of colon definitions. The definition is *short*! This aids immeasurably in testing assembly language routines. **3*** will appear in the dictionary just like it did when defined as a colon definition. All the other words that call **3*** will work the same. They won't know the difference. Also, **3*** can still be tested directly from the keyboard. If you have to deal with assembly language, this is the way to do it.

For the curious I'll explain the parts of the code definition above. It is written for Pygmy Forth, which keeps the top of the data stack in register BX. The assembly mnemonics end in commas. You can think of the comma as marking the end of a phrase, thus **BX AX MOV,** is one instruction. The source register and destination register are arranged left to right as god and Motorola intended. Thus, the first instruction copies the top stack item to the AX register, because we'll need it in a minute. Then, the **BX BX ADD,** instruction doubles the top stack item. This is the same as multiplying it by 2. The **AX BX ADD,** instruction adds the original value to BX, giving just what we want, 3 times the original value. Since the whole word consumes one stack item and returns one stack item, and since Pygmy keeps that stack item in register BX, the final result is right where we want it in the top stack item. The last "instruction" **NXT,** is actually an assembly language macro defined in Pygmy to lay down the code that will move to the next word to be executed.

The assembly language will differ from Forth to Forth and especially from processor to processor. It is less portable and more trouble than high-level Forth, but is *very easy* compared to most other methods of dealing with assembly language! If you want to do everything in assembly, you could think of Forth as an interactive assembly language subroutine manager, allowing you to test each little subroutine from the keyboard.

You Try It

Practice makes perfect. Try reading some Forth and see if it makes more sense to you now. If you want a Forth to practice with on the IBM PC/XT/AT/386 you can get my shareware Pygmy Forth version 1.3 from the Forth Interest Group (FIG) at (408) 277-0668 or from finer bulletin boards everywhere. If you get it from FIG be sure to ask for a free quick reference card as well. If you have specific questions, you can reach me c/o *TCJ* or on GEnie as F.SERGEANT, and I'll try to answer directly or via *TCJ* or both.●

"Never Tangle with Any Creature That Has More Teeth Than the Osmond Family" — anonymous

The Z-System Corner

The Trenton Computer Festival

By Jay Sage

Announcements

The announcements for this time are about some exciting price reductions. A small band of programmers has long enjoyed the help of the wonderful DSD (Dynamic Screen Debugger) in getting the bugs out of our code. The $130 price tag, however, was a deterrent to many. I recently located and spoke with the author, John Otken, and suggested that there would be a great advantage in reducing the price. More people would get to take advantage of his superb program, and he would actually make more money because of the increased sales. He agreed to my suggested price of $50!

I hope that many more of you will take advantage of this opportunity. DSD is not only a powerful aid in debugging problems with code; it is also a wonderful way to learn about how programs work. The full-screen display allows one to see everything that is happening during program execution. One sees a section of disassembled source code, the contents of the CPU registers, the contents of the stack, and two blocks of memory. In all cases, symbolic as well as numeric data can be seen, and all user entries can make use of defined symbols.

There have been two price reductions of interest to owners of SB180 computers. XBIOS has been reduced from $75 to $50, as a reflection of the fact that it is no longer actively supported. We have also acquired the remaining stock of the excellent Electronic Technical Services ETSIO180+ board. Its low power CMOS design includes a battery-backed real-time clock, two high speed (115.2 kbps) serial ports, 24 bits of parallel I/O, and an SCSI interface for hard disk drives. It is fully supported by XBIOS (which is required to run the board). I use the ETSIO180+ on my own computer and have been very satisfied with it. The original price was $280; we will sell the remaining stock (currently about six boards) for just $100 each (or $145 including XBIOS).

The Trenton Fiascos

The main contents of my column for this month were originally intended for publication in the printed proceedings for this year's Trenton Computer Festival. Although *our* session there was again a great success (because of the speakers), the organization of TCF has deteriorated alarmingly over the past few years.

When I originally sent in my speaker form for TCF, I checked the box indicating that I would contribute a written piece. Shortly before the deadline, when they had not received my submission, I was called by Sol Libes, who runs the conference, asking if I was going to write something. I told him that I was still interested if there was time. He said that it could be accepted if I got it in by the end of the next weekend. We agreed that I would upload it to the board run by the Amateur Computer Group of New Jersey. Once it was there, I was to call Sol.

I worked extremely hard for a couple of days, staying up until 3 am to get it finished and uploaded to the ACGNJ board. As instructed, I placed a voice call to Sol. He was not there, so I left a message on his answering machine. When he did not call back, I called several more times, leaving messages each time. Finally he returned my call, and we agreed that he would call me again if there was any problem with the file I had left for him. I never heard anything further.

You can imagine my dismay when I learned during the festival that my article had not been included. I still have no idea why, as no one has had the courtesy to contact me. Was its omission deliberate? I'm inclined to think it was administrative incompetence, since most other aspects of the conference organization were also handled poorly.

On my speaker registration form I had checked boxes indicating the equipment that I wanted them to supply in our meeting room. My speaker confirmation form indicated that the information had been recorded, but I telephoned the program coordinator the day before I drove down just to be absolutely sure. Do you think the equipment was there? No. And do you think that anyone at TCF knew what to do about it? No. We had to struggle through a couple of talks before, almost by accident, we stumbled upon the people who had the equipment (no

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR command processor, his ARUNZ alias processor and ZFILER, a "point-and-shoot" shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (MABOS on PC Pursuit, 8796 on Starlink, pw=DDT). He can also be reached by voice at 617-965-3552 (between 11 p.m. and midnight is a good time to find him at home) or by mail at 1435 Centre Street, Newton Centre, MA 02159. Jay is now the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail, or chatted with live at the Wednesday real-time conferences (10 p.m. Eastern time).

In real life, Jay is a physicist at MIT, where is tries to invent devices and circuits that use analog computation to solve problems in signal, image and information processing. His recent interests include artificial neural networks and superconducting electronics. He can be reached at work via Internet as SAGE@LL.MIT.EDU.

one had told them we needed it).

I had hoped that the following material would be read by an audience that might include many people with CP/M computers who did not know that magazines like *TCJ* and advances like Z-System existed. [Ed: Thanks for the thought anyway, Jay.] So that my effort on this composition was not totally wasted, I am presenting it here.

CP/M Is Not Dead!

I'm sure you all know the famous Mark Twain anecdote. Somehow the newspapers picked up a story that he had died. At his next public appearance he delighted in relating that his reported death was, as he put it, "greatly exaggerated." The rumored death of CP/M, the granddaddy of microcomputer operating systems, is similarly exaggerated!

Unfortunately, the erroneous impression about CP/M is much harder to correct. Were it simply the result of false reports in the media, one could bring the truth to the media's attention and have them publish a correction. Regrettably, the belief that CP/M is dead arises via a mechanism that cannot be used to correct the view.

I will try here to give you some impression of the vitality that continues to flourish in the CP/M world, particularly in connection with the development of the Z-System, a highly advanced successor operating system to CP/M. While maintaining total compatibility with CP/M and its software base, the Z-System brings concepts and features that are as advanced as can be found on any computer anywhere. Nothing can give a 4-MHz Z80 with 64 kilobytes of memory the raw computing horsepower of a 33-MHz 80386 microprocessor with several megabytes of memory, but the Z-System can give an old CP/M computer a surprisingly powerful user interface and can turn it into a fun, educational, and productive machine.

At the end of this article, I have listed three magazines with strong CP/M support. By subscribing to them you will be able to learn about sources for public-domain programs and the names and addresses of vendors who actively support 8-bit software and hardware products. You are also welcome to call me.

Why Do People Think CP/M Is Dead?

The community did not learn of the death of CP/M, as with Mark Twain, by reading a report; they inferred it from personal experience. The mainline computer magazines, such as Byte, gradually carried less and less news about CP/M until eventually they carried none. Why did they stop? Basically because the volume of interest shifted naturally enough to MS-DOS and the newer hardware. There was no longer much activity in CP/M hardware development or sales, and almost all mainline software houses stopped developing new CP/M programs. Soon most of them stopped even offering their old CP/M programs for sale. Some, like Borland, have even gone so far as to deny that CP/M products they once offered were *ever* offered by their company! (I swear I remember a Borland Turbo Modula 2, but many people tell me that when they contact Borland they are *assured* that Borland *never* offered that product.)

This situation notwithstanding, a very high level of CP/M activity did continue, but not in the spotlight. There were smaller, specialty magazines that continued to focus on hobbyist computing, and there were also smaller software vendors who continued to support CP/M with great enthusiasm. For veteran CP/M users there was some chance of their learning about these alternate sources of support before the mainline sources shut the door on CP/M, but even many of the veterans lost touch.

For new owners of CP/M computers—and though it might surprise you, there are many—the situation is worse. Typically, they have inherited a retired CP/M computer from a family member, friend, user group, or even a stranger. When they try to contact the sources for CP/M information and support listed in their old documentation, they find them to be either out of business totally or at least out of the CP/M business. And these days the sales representatives at the software houses have often never even heard of CP/M! No wonder these new CP/M computer owners conclude that CP/M is dead.

The Real Story

It is natural that the major focus of interest in computers will continually shift toward the latest, most advanced, and most widely sold hardware. Nevertheless, a tremendous level of activity still exists in the CP/M world, most of it centered on the further development of the Z-System.

Why would someone still be interested in CP/M? The special spirit that persists in the CP/M world is one important factor. As in the Unix world, there is a very strong sense of community and sharing. This includes a well established tradition of public-domain software, that is, programs written specifically for the benefit of the general community and shared totally with that community. By contrast, the MS-DOS world has what they misleadingly call "shareware." These programs are distributed for free, but users are told that they must send money to the author if they continue to use them.

A second attraction in the CP/M world is the intellectual challenges and rewards it offers. The relative simplicity of the Z80 microprocessor and the CP/M operating system allow easy entry to new programmers. Although extremely rare in the DOS world, it is nearly universal in the CP/M world that programs are published with source code. This has several far-reaching consequences. In terms of quality, it means that the development of a program is not limited to the conceptual and coding ability of a single author. Others in the community can take that source code, make improvements to the program, and return the results to the community. Many of the most important CP/M programs have a very long history of development involving many authors.

Reading source code is also an indispensable vehicle for learning about how a computer system works. New programmers can quickly learn coding techniques and apply them to new applications. Amazingly, source code is available not only for the numerous utility programs but also for at least some versions of all parts of the operating system as well.

The seminal event in the survival of CP/M took place almost exactly a decade ago. Richard Conn, eager to make his own contribution to a community that he felt had helped him so much, spearheaded a group effort to write replacement code for the Console Command Processor (CCP) portion of the CP/M operating system. The result, named ZCPR for Z80 Command Processor Replacement, was released to the public in February, 1982.

Not much later, I discovered this much improved command processor

and installed it on my CP/M computer. Then I did something even more significant: I looked at the source code. To my amazement, I discovered that the operating system, that "holy of holies," is nothing more than a computer program, rather like any other, and that I, albeit untrained in systems programming, could not only understand it but experiment with changing it! Discoveries like this from reading source code continue to inspire new CP/M hobbyist programmers, and computer bulletin boards and user group meetings continue to be scenes of intense discussion.

The Features of Z-System

ZCPR command processor has gone through several stages of evolution, and the same approach has now been applied as well to the Basic Disk Operating System (BDOS) component of the CP/M operating system. The combined result is now referred to as the Z-System, and it represents a remarkable point in the development of microcomputer operating systems. It has many, many features inspired by minicomputer and mainframe operating systems and some not available even on those powerful machines. When I use my DOS 386 machine at work, there are many Z-System features that I sorely miss, and I am constantly amazed at their absence from an operating system whose code is bigger than the entire memory address space on a CP/M machine!

For the most part, the original Digital Research CP/M accomplished in admirable fashion the essential functions it was required to perform: to run a few resident commands and to load external commands from disk. However, it devoted little attention to the user interface and did not provide many services to make the operator's life easy. To be fair, of course, CP/M was born in the days when 16K of memory cost about $500 (in 1970s dollars, no less) and occupied an entire S-100 card, roughly the size of the motherboard of a modern DOS machine.

The central goal of the Z-System since the beginning has been to make it easier and more convenient to operate the computer. My own ideal has been to have the computer perform all tasks that it can handle on its own and to leave to the user only those tasks that a computer cannot figure out by itself. In other words, the computer should take care of all the routine matters; the human operator should handle only the thinking. I will now run through a short summary of Z-System features and try to indicate how they make the operator's life easier.

User Area Access

CP/M introduced the concept of disk "user" areas, which allowed the operating system to group files into separate logical directories. Unfortunately, CP/M provided no practical way to access files across user areas, which made them almost useless.

Z-System makes it very easy and convenient to organize your files. Where CP/M allowed only a drive prefix in file specifications (e.g., B:TEXT.DOC), Z-System allows drive and/or user number prefixes (e.g., A3:WORK.WS) so that files in other user areas as well as other drives can be referenced directly. In addition, Z-System allows meaningful names (similar to DOS subdirectory names) to be assigned to drive/user areas (so we might have LETTERS:JOE.WS). This provides an interface that is far more suitable to the way people think and remember.

Terminal Independence and the Environment

While some would argue that the DOS hardware and software standards established by IBM's market dominance have resulted in an enforced mediocrity, there is no doubt that having a single environment in which to operate makes life much easier for applications programmers. Programs for DOS generally work right out of the box on any IBM-compatible computer. Configuration is required only for fine-tuning.

CP/M, on the other hand, was designed to allow programs to run on an extremely wide variety of hardware. In those days, "personal" computer took on a different meaning—each person designed and built his own hardware. CP/M could be made to work with all of them, but elaborate configuration procedures were generally required, especially to match programs to the particular terminal used. To this day, we still have to deal with this hardware diversity.

What CP/M could have provided, but didn't, was a means for conveying to application programs information about the operating environment. Z-System has several modules that afford such communication. An area called the environment descriptor (ENV) contains information about the system configuration. Another system area called the message buffer (MSG) stores information that one program can leave for use by another program that runs later.

Part of the ENV is a section called the TCAP or Terminal-CAPability descriptor (if you know Unix, you will recognize this). When the appropriate TCAP has been loaded, a Z-System program will automatically use the right video control codes for that terminal.

Command Processing Enhancements

Under CP/M, you had to specify where the COM file to be run was located (otherwise the current drive was assumed). This is a perfect example of something that a computer can easily be smart enough to do for you, and Z-System does. As with modern versions of DOS (which took many years to catch on to this Z-System feature), you specify a list of directory areas that the operating system will scan for a requested COM file. Unlike DOS, Z-System does not insist that the current directory be first in the path; it can appear wherever you want it to, or not at all.

With Z-System one is also no longer limited to issuing commands one at a time (DOS has been even slower to catch on to this). A single line of command input can contain a whole sequence of commands. As a result, you do not have to interrupt your thinking to wait for one command to finish before you can specify the second and subsequent steps in a process. You can work out a strategy for what you want to accomplish and issue all the commands at once, before you forget or get confused.

Many oft-repeated computational tasks involve sequences of commands (e.g., editing, spell-checking, printing). In such cases, the Z-System alias facility (similar in some ways to CP/M's submit or DOS's batch operations but far more flexible—more akin to Unix scripts) can be used to define a new command name, which, when invoked, performs the entire sequence. This saves the user a lot of typing but more importantly eliminates the need to remember exactly what the sequence

is. Basically, you solve the problem once and put the solution into an alias script. From then on, the computer is smart enough to take care of the complex details for you.

Conditional Command Execution

There is only so much one can accomplish on a computer (or in life) without making decisions. Have you ever seen a programming language with no ability to perform tests and act in different ways depending on the results? Flow control (IF/ELSE/ENDIF) is unique to the Z-System command processor. Other operating systems that offer flow control at all limit it to operation inside a batch or script language. A special set of Z-System commands can test a wide range of conditions, and the tests can be nested to a depth of eight levels.

Command Processor Shells

If you do not want to deal with the operating system at the command level or if you want to have a command processor with extended features, the Z-System shell facility allows you to install substitute user interfaces of your own choice at will. They can even be nested within each other.

Shells come in two common varieties: menu shells and history shells. The menu interfaces allow the user to pick tasks with single keystrokes and have the shell program generate the complex sequences of commands required to perform those tasks. The menu system shields the user from complexity, saves typing, and greatly reduces the chance of error.

History shells are enhanced command processors that remember your commands and allow you to recall and edit previous command lines. I wish the Apollo Domain minicomputer system I use at work (not to mention my DOS computer) had a history shell as nice as Z-System's LSH. It works like a wordprocessor on the command history, allowing searching and extensive editing.

What If You Make a Mistake

This is one of the other areas in which most operating systems behave in an abominably primitive manner. When you issue a command that cannot be performed, they just issue an error message and then dump you back to square one. Often you are not even told what sort of error occurred (consider DOS's wonderfully helpful "bad command" message).

The Z-System behaves in a civilized manner under these circumstances. When an error occurs, the command processor turns the bad command line over to a user-specified error handler. The most sophisticated error handlers allow the operator to edit the command and thus recover easily from typing mistakes.

The system environment even contains an error type, which the error handler can use to give you more specific information about what went wrong. It may be the familiar error of a COM file that could not be found, but there are many other possible causes for the difficulty. A file that you specified as an argument might not have been found (e.g., "TYPE FILENAM" when you meant "TYPE FILENAME"), or you may have specified an ambiguous file name to a program that cannot accept one (e.g., "TYPE *.DOC").

System Security

Like minicomputer and mainframe operating systems, the Z-System is a secure operating system. This means that it has mechanisms for limiting what any particular user can do or get access to. Commands that perform dangerous operations (such as erasing, copying, or renaming files) can be disabled when ordinary users are operating the system but enabled when a privileged user is at work. Areas of your disk can be restricted from access for storage of confidential or other sensitive information. These security features come in very handy in the implementation of a remote access system or bulletin board.

Summary

To sum it up, the goal of the Z-System is to provide an operating environment that can be tailored extensively to user preferences and that can be made to handle on its own and automatically as many computational details as it can, leaving the user free to concentrate solely on those aspects of computer operation that require human intelligence.

Z-System, The State of the CP/M Art

Now that you've seen the remarkable features of the Z-System, you might be curious about how it can be installed on your CP/M computer. In this regard, the most spectacular step in the evolution of Z-System took place about three years ago. At that time some major programming breakthroughs made it possible to configure and install the Z-System as if it were an application program running on the CP/M computer. Before that, you really had to be a rather skilled assembly language programmer to get Z-System running.

The new Z-System comes in two flavors: NZCOM for computers that use version 2.2 of CP/M and Z3PLUS for computers running CP/M-Plus. Joe Wright, Bridger Mitchell, and I were the main developers.

NZCOM and Z3PLUS embody, I believe, some of the most exciting and remarkable developments in the history of microcomputer operating systems. With all the microcomputers I have had experience with, the operating system has been a static entity. You boot up the computer, and there you have the operating system, fixed and immutable. Few computers offer more than one operating system. With those that do, changing the operating system usually requires rebooting and starting over. And never do you, the user, get to define the characteristics of the operating system. You just take what the manufacturer decides to give you.

With NZCOM and Z3PLUS the operating system becomes a flexible tool just like an application program. You can change operating system configurations at any time, even right in the middle of a multiple command line sequence. You can do it manually, or alias scripts can do it automatically in response to conditions in the system!

You can change the whole operating system or just a part of it. Would you like a different command processor? No problem. With a simple command, NZCOM or Z3PLUS will load another one. No assembly or configuration is required. If you want to experiment with a new disk operating system (BDOS), NZCOM can load a new one in a jiffy. This makes for a whole new world of flexibility and adaptability, learning and experimentation.

Do you need more memory to run a big application program? Fine. While that application is running, just load a small operating system without some of the bells and whistles. Then, after that task is finished, go back to the big system with named directories, lots of resident commands, or special input/

See Z Corner, page 48

PMATE/ZMATE MACROS

4. "Mother of All Macros"

By Clif Kinne

Before I get engrossed in this issue's subject matter, let me ask once more for your feedback. This has been a one-way street so far. True, I have managed to clean up and enhance my MATE in the process of preparing these columns, but I really miss the stimulus of your responses. Who is going to call oversights to my attention if not you? (Did anyone notice the missing lines at the ends of 3 listings in column 2? See corrections in *TCJ* #50. Also in that issue, I just discovered I failed to precede instances of the GoBack macro, .^G, with its argument, @7. Please mark your copy with that correction.)

Mainly, however, I am sure there are macro ideas out there that I've never thought of. Please share them with others, through this column. And, for some of you, I may have raised more questions than I have answered. Please, let me know.

In addition, I should very much like to know if there are readers using each of the variations of PMATE: MATE, ZMATE, and PCMATE, and to what extent.

Buffers as Macros

You have noted, I am sure, that all of the macros offered so far have been for the permanent macro area. This time we turn our attention to buffers as macros.

Many times we write macros for use with particular files in one user, or directory, area. It seems wasteful to clutter up the permanent macro area with such specialized macros. Using up the limited supply of names available for permanent macros is of even greater consequence. Thus it makes sense to let such macros reside as files in the relevant user area and be called into a buffer for execution when needed.

However, since most macros are no more than a few hundred bytes long (usually much less), it also makes sense to collect most of those macros in a single file for each user area. This, then, leaves us with the problem of executing a particular one of those macros after they are loaded. My solution to this problem is this month's macro.

The names for the macros in previous columns came easily to mind, and I thought them fairly apt. But, I must tell you, I didn't have any idea of a name for this one until Saddam came along. I'd like to think that this is as effective in managing macros as the "Mother of All Battles" was in managing Saddam.

The Mother Macro

Before calling your attention to the coding in Listings 1 and 2, I want to describe the external nature of the macro: a) some of its features and b) how it fits into the overall scheme of my PMATE.

Salient Features.

Again I am indebted to Jay Sage. First for pointing out that this macro introduces two novel techniques:

1. A macro which modifies its own code in accordance with external input supplied by the user. In this case, the target label, s, of a Jump command, Js, is changed to a character typed by the user.
2. The use of scrolling when displaying a menu, to remove extraneous material from the screen.

And second, for prodding me into making the Mother macro independent of the nature of the menu. This turned out to have several beneficial side effects.

Integration into PMATE.

As you might expect, the effectiveness of this macro is at least as dependent on how it is integrated into your PMATE system as it is on the coding of the macro. I shall describe how I have done it, and offer help on variations only if asked:

In each user area having regularly edited text files there is a file called MACS.MAT. Each such file holds macros specific to it's User Area, along with some general-purpose macros.

The autoexec macro (the permanent macro which is executed when PMATE is brought up) loads MACS.MAT into buffer 9.

Buffer 9 is executed by pressing Function Key F9.

Having lived so closely to this macro during its many years of evolution, I fear that I may underestimate the difficulty of grasping how it works. In an effort to ease such difficulty, I am presenting it in stages: first an implementation of just the minimum features necessary to the macro management; then with added frills to make it easier to use. (This was not, of course, the way it evolved, but, believe me, you don't want to go through the convolutions of its actual development!

The MACS.MAT File.

Figure 1 depicts the general nature and appearance of a minimal MACS.MAT file (not including the line numbers on the right, which are for

Clif Kinne is a retired computer designer. He cut his teeth on vacuum tube and acoustic delay line machines in the fifties, made the transition to transistors and magnetic cores in the sixties, left the field to his children in the seventies, and tried, vainly, to catch back up with them in the eighties. He can be reached by voice at 617-444-9055, or via a message on Jay's BBS, 617-965-7259. His address is 159 Dedham Ave., Needham, MA 02192

Listing 1. Code for the minimal Mother Macro.

```
        ;MiniMom                                        38 bytes

;       FUNCTIONAL SPECIFICATION:

;       1.  Moves to Buffer 9
;           a.  Displays the labeled "Children" macros.
;           b.  Prompts user to type a macro label.
;           c.  Replaces the final character in MiniMom
;               macro with the character typed.
;       2.  Returns to home buffer.
;           a.  Jumps to label chosen
;           b.  Executes that "Child" macro.

        CODE:
@BV7            ;Save home buffer in V7.                    1
B9E             ;Go to buffer 9.                            2
A               ;To top.                                    3
2{SEJ$}         ;Move past 2nd EJ in this macro.  (E is     4
                ; added to provide a more nearly unique
                ; string to search for than 'J' alone.)
GWhich macro?$  ;Prompt user to enter label letter of       5
                ;  desired Child macro.
@KR             ;Overwrite the target byte of this          6
                ;  J instruction  with the key typed.
@7.^G           ;Return to home buffer (MATE).              7
EJs             ;Jump to selected label and execute that    8
                ;  macro.
```

reference use in this discussion).

Line #1 is the minimal Mother Macro, "MiniMom". Following that are a few "Child" macros, some truncated to conserve space. The labels for these "Children" can be any printable characters in any order you like.

Embellishment of the minimal MACS.MAT file to that of Figure 2 is explained as follows:

1. The Mother Macro does not have to be the first line of MACS.MAT, just the first executable code. Thus, we can put an embedded filename on the top line and skip a line, for appearance sake, before Mother. The top line of figure 2 also includes recent revision dates and a reminder of its user area.

2. Note that it doesn't matter what is between the final command, Js, of Mother and the first following label. I have taken advantage of that to put there a 21-line menu (22-line for 25-line screens) that will neatly fill the screen when buffer 9 is called. Note that I have also added a column to remind the user of buffers used.

You could, instead, put in any message and instructions to the user that you like. Just don't make a colon the first character of any line!

3. You notice that Mother is spread over 2 lines in figure 2. That is because it includes a search for a 2-byte string: a CR followed by a Colon. That command necessarily inserts a carriage return into the macro.

4. In figure 2, I have included my full menu, but only about half of the actual macros. I thought you might be interested in some that are helping me to write these columns. In macro :E you will have to figure out which of the dollar signs are Escapes and which are real dollar signs.

The Commented Macros, Listings 1 & 2.

In both of these listings we have to get the cursor on the target character of the final J command so that we can replace it with the label selected by the user. If we just execute the command, ASJ$, we shall find the cursor on the ESC following the J in the command we have just executed, or an

Figure 1. Typical Implementation of a Minimal MACS.MAT file

```
@BV7B7E A 2{SEJ}GWhich macro?$ @KRB@7EJs                    1
                                                            2
:2 B8E.^B@S{XXXIFORMS.MAT$}JQ   ;Illustrates handling       3
       ;of large macros like FORMS.MAT.                     4
                                                            5
:6[ES  $@E_-3M@T=(13){MS^N $^}@T="?!(@T="!){GEnd of sent?$ 6
  .^Y@S'^MJX}@T=".{M}:X2M[@T=" '_gDelete?$.^Y@S'_D]]JQ     7
                                                            8
:9 .^PFind:$@8'{#B2C}-L3K[LES;$@E_E0S^A@2$@E^               9
   G(Q)uit$"Q.^A@S]0GDone!$40QDJQ                           10
                                                            11
:B [@T=0_@T=13!(@T=";){K^}$[@T=32'_D]                       12
   E1S;$@E^-D[-M@T=32!(@T=9)'_D]MK]JQ                       13
                                                            14
:Q QBB@7E^X                                                 15
```

Listing 2. "Mother of All Macros"

```
^X9     ;Mother                                         69 bytes

;       FUNCTIONAL SPECIFICATION:

;       Manages a collection of macros in Buffer 9.
;       1.  Displays a menu of those macros.
;       2.  Responds to the user's keypress by beeping
;               if no macro label corresponds to that key;
;               else jumps to that label.
;       3.  Executes that macro.
;       4.  Jumps to the common cleanup code at label :Q.

;       VARIABLES USED: V7 saves home buffer.

;       BUFFERS USED:   B9 holds this Mother macro and the
;                           "Children" thereof.

;       SUBROUTINES:                    USING:
;               .^R     Restore         .^G  GoBack
;               .^S     SaveEnv         .^G  GoBack

;       USAGE:  This makes most sense if the macro, .9,
;               which executes buffer 9 as a macro, is in
;               the permanent macro area and can be invoked
;               as an Instant Command by 1 or 2 keystrokes,
;               preferably function key F9, if available.

;       CODE:

@BV7            ;Save home buffer in V7.                    1
B9E             ;Go to Buffer 9                             2
.^S             ;Save environment of Buffer 9               3
:Y              ;Label for restart if no macro found.       4
A2{SRJ$}        ;Move to Jump target at end of this macro   5
                ; (assumes no other preceding RJ strings
T               ;Tag this target character location.        6
L               ;Move to first line after Mother macro.     7
@LQJ            ;Scroll Mother macro out of view.           8
G$              ;Wait for keypress.                         9
[          ;Search for macro label.                         10
  E             ;  Disable error messages.                  11
  S:$           ;    Search for a colon in column 0.        12
  @E{           ;    IF no label is equal to key struck,    13
    QB          ;      THEN beep and                        14
    JY          ;      redisplay menu.                      15
    }           ;    END IF                                 16
  @T!32=(@K!32) ;  IF character at cursor = key struck,     17
]               ;  THEN escape loop; ELSE loop again.       18
#               ;Return to Jump-target character and        19
@KR             ;replace it with keystroke character.       20
.^R             ;Restore buffer 9 environment.              21
@7.^G           ;Return to home buffer (for MATE),          22
;B@7E           ;(Alternative for PCMATE or ZMATE.)
QR              ; and redraw it.  In the executable
                ; version there must be no character
                ; between this R and the following J.
Js              ;Jump to selected macro for execution.      24
```

earlier J in the macro, if any.

To get around these difficulties, I have elected to:

1. Make the search target the 2-character string: J with its preceding character.

2. If the preceding character is inappropriate, insert arbitrarily an E, which should have no side effect.

3. Search for the second instance of that target string with A2{SEJ$} (or A2{SRJ$} for Listing 2).

The two major enhancements to the Mother macro, detailed in Listing 2, are:

a. The bracketed loop checks if there is, indeed, a label corresponding to the user's keypress. If not, it simply beeps and waits for another try, instead of letting PMATE complain: "STRING NOT FOUND".

b. The code, L@LQJ, scrolls the Mother macro and any preceding lines out of sight and leaves just the 21- or 22-line menu visible. (Since the menu communicates with the user, there is no need for a command-line message.)

Application Notes

1. Note the distinct similarity to Instant Commands. Any of the macros can be executed with 2 keystrokes: F9 plus the menu item key. However, you don't have to remember a macro's name in order to invoke it. In fact, you don't even have to remember what macros are available.

2. Of course, when MACS.MAT is in buffer 9, it reduces memory available for your working text. Sometimes you will want to delete buffer 9 for that reason. In any case, this is a good reason not to include any Children that run to a kilobyte or more.

3. A way to handle such large macros is illustrated by line 3 in Listing 1. That makes FORMS.MAT, which runs to 5 kilobytes, readily available but not in memory until really needed. If you recall my column 2 (TCJ 49), my Buffer 8 is executable by function key, F8. So, after executing macro 1 of Buffer 9,you will have to type F8.

4. My macro labels (menu item numbers) can be thought of as hex digits, since there is just about room for 16 of them if no more than one or two use two lines. Of course, as indicated above, you can use any other scheme you like for macro labels and menu item designators.

5. Note that we save the environment (.^S) after entering Buffer 9 and restore it (.^R) before returning to the home buffer. This is advantageous when debugging a Child macro. After trying the Child out and returning to Buffer 9 for more editing, we find the cursor just where we left it.

6. To invoke one of these Child macros from another macro is a bit awkward, but it can be done. For example: Say you want to call :3, having your home buffer already stored in V7. With PCMATE, you can make use of the ..n command, which starts executing Buffer n at the cursor position:

```
B9E A S:3$ B@7E ..9
```

should do it. For MATE or ZMATE, you can move macro :3 to another buffer for execution:

```
B9E A S:3$ T S:4 #B2C @7.^G .2                ;For
MATE.

or          B9E A S:3$ T S:4 #B2C B@7E  .2 ;For ZMATE.
```

is a possibility. Of course, you should check that you don't have ":3" or ":4" in a comment somewhere ahead of the labels. If you do, you could search from the bottom up, maybe. Just change A S:3 to Z -S:3.

7. Some macros you don't always want to run from the beginning. You can use this Mother macro technique to invite optional starting at any one of several points in a macro. Simply write the menu describing those points, and label them accordingly.

For Next Time

It's just possible I may skip the next issue. Without some reader feedback, the strain of just deciding where to go from here is becoming unbearable. ●

[Three corrections from last issue:

1. Listing 1, line 4 should read "@A%" instead of "@0%".

2. The paragraph on the bottom right column of the first page of Clif's article starting with "Since any contents of this buffer are wiped out" should have been preceded with "0", which is the buffer number in use.

3. Listing 2, line 35 begins with an extraneous character that evidently was picked up during electronic transfer and not noticed. Please disregard the "é".

Again, our apologies for any inconvience. Editor]

Figure 2. Contents of Full MACS.MAT file in Buffer 9.

```
;MACS.MAT    5-3'91 4-10'91 4-7'91 3-13'91          TCJ.MAT  1
                                                              2
@BV7B9E.^S:YA2{SRJ$}TL@LQJG$[ES                               3
:$@E{QBJY}@T!32=(@K!32)]#@KR.^RB@7EQRJQ                       4
                                                              5
             - - ENTER NUMBER OF MACRO CHOICE - -             6
                                                              7
Q.  Quit                                        Buffers used  8
                                                              9
0.  Set TABs.                                                10
1.  Get date.                                                11
2.  Get telephone number(s).                          0,3    12
3.  Move table entries around.                        5,7,8  13
4.  Compare T buffer with buffer 1, line at a time.   0,1,2  14
     Comparison ends at first ESC in either line7            15
5.  Get list of unused macro names.                   1,2    16
6.  Remove redundant spaces.                                 17
7.  List variables.                                   0      18
8.  Number macro lines for TCJ.                              19
9.  Search for a string to left of semicolon.                20
A.  Make soft CRs hard.                                      21
B.  Compress commented macro                                 22
C.  Get buffer size (excl. top line).                        23
D.  Number all lines on right.                               24
E.  Change escapes to dollar signs.                          25
F.  Change Ctrl-letter macros to caret,letter strings.       26
                                                             27
:6 [ES  $@E_-3M@T=(13){MS^N $^}@T="?!(@T="!)                 28
   {GEnd of sentence?$ .^Y@S'^MJX}@T=".{M}                   29
   :X2M[@T=" '_GDelete?$.^Y@S'_D]]JQ                         30
:8 GIs there a blank line after last line?$                  31
   .^Y@S'{0GMake one$30QDJQ} .^S1V1                          32
  GMove cursor to first line to number. Then type ENTER.$    33
  -L [L@T=0_@T=13{QBGDone?$.^Y@S_^}$[@T=32!(@T=9)'_M]        34
 @T=";{L-M[@X<57_-D]^}57QX@1<10{32I}@1\VA1[@T=13_D]].^RJQ    35
:9 .^PFind:$@8'{#B2C}-L3K[LES;$@E_E0S^A@2$@E^                36
   G(Q)uit$"Q.^A@S]0GDone!$40QDJQ                            37
:B [@T=0_@T=13!(@T=";){K^}$[@T=32'_D]                        38
   E1S;$@E^-D [-M@T=32!(@T=9)'_D]MK]JQ                       39
:C AL@C,Z@C-@S\GType any key$0KJQ                            40
:D 1V11[@T=0_@T=13{GDone?$.^Y@S_}                            41
   58QX@11<10{32I}@11\VA11[@T=13_D]L]JQ                      42
:E [S^L$$GChange$.^Y@S{-M"$R}]JQ                             43
:F[S.$@T=9^@T=13^@T="Q_@T<32{GChange?$.^Y@S{"^I@T+64R}}]JQ44
:Q QBB@7E^X                                                  45
```

Corner, from page 64
spent hunting down parts.

With systems coming along like these it backs up what I said last time about not needing to teach the hardware side of electronics anymore. Pretty soon even embedded systems will be plug and play operations.

EForth68K

I have been working on EForth and got it running on my Sage system. The EForth is not much of a system, but if you want a quick Forth on a new system here is the product. It can be downloaded from the GEnie Forth section and assembled with Microsoft's MASM. Now I am not one to say much for Microsoft's products, but their assembler does work and as I will point out does pretty good over what Motorola assemblers can do.

So what I did was download one of the EForths for the PC and played with it. Then I got one for the 68K from GEnie. Now it is still in the MASM format, just the 68K code is entered as defined words. This means all the hand coding has been done for you. What will need changing is the terminal input and output ports or addresses. To test it out I just replaced the I/O subroutine calls with ones to my hardware, using DOS's DEBUG. You can edit the S records with an word-processor or patch the binary code supplied. I tried reassembling and combining the code properly but found the special routines crashed.

The resulting product is pretty good, but you need to use MASM in order to get all the macros to work properly. I found this out the hard way when I ported the code to several 68K assemblers. I used mostly the Avocet and Microtek assemblers. Each has their own problem which must be overcome. In the Avocet you use a macro preprocessor which I like as you will see later. The Microtek you assemble like the MASM, all in one pass.

Way back when, Motorola used a pretty limited set of characters in their assemblers, and especially their macro assemblers. MASM has no limits and will assemble all the macros as supplied. Both 68K assemblers will choke badly on the macros. I learned first hand why most people metacompile Forths. Forth's special use of characters just drives the assemblers crazy. With Avocet the preprocessor allows you to go over the code after the macro and correct any of the failures (I have learned to like that feature a lot!). There are several characters that can not be used in macros at all and those words will have to be done by hand coding, period.

I now have a working (with a few minor bugs) 68K version for 68K assemblers. A number of problems were solved by doing away with any macros. In MASM you can equate locations over and over again. The Microtek assembler has limited number of times you can equate things or set the origin. We ran into this problem earlier with their use of strings for the section command. I think you can have 256 different sections each with a 32 character string name. That is pretty nice except it drives us crazy when we try to do math inside a macro to determine which section code should go into. Now this math inside a macro is not my idea and I want no part of it, but my company's code has tons of this sort of nonsense and the Microtek will not work using it. You have to manually set each section name and ORG statement and do not exceed 256 of both.

"In MASM you can equate locations over and over again. The Microtek assembler has limited number of times you can equate things or set the origin."

The EForth uses a macro that re-ORGs the word as it is being defined. They do this to get a separate dictionary in upper memory going downward, while the code section starts in lower memory going upward. I solved the problem by just having two separate code sections, with the dictionary starting from a pointer that is calculated as each word is defined. The original code was about 40K in length (text words) and after removing the macros and creating a separate word linked list, you have about 140K of assembly code. So yes the macro listing is short until expanded (about 300K expanded), but without expanding you have little knowledge of what the actual code is.

In my code, what you see is what will be assembled. It also makes how the assembler actually does things a lot more clear. This last point I feel is important as the main reason for EForth was to provide a quick and simple learning process for new people to Forth and to bring other systems on line quickly. Now patching the existing code is probably the fastest and easiest way to get up and running. My 68K code will get you there, but you do need to know about 68K assemblers and such. The macros just hide too much of what is going on from the programmer and so little is learned. The whole linked list was made inside the macro where as mine is listed code showing the links, names, counts, and flags. Make it simple if you want people to learn from it and progress.

Hopefully by the time you read this my code will be bullet-proof and on GEnie. I discovered the 68K assemblers do -1 differently than MASM and some flags are handled differently as well. I have found most of the differences, but one of them still eludes me. I have been forced to break down and do a bit-wise comparison of the output to find the problem. Needless to say it takes time as some difference are acceptable, some are not.

"The macro listing is short until expanded, but without expanding you have little knowledge of what the actual code is."

Minix

I have talked to fellow workers using Minix 1.5 on a MacIntosh and they seem very pleased. I have version 1.3. There is a discount for upgrading and I will do it later for the PC. My Atari ST however just has 1 meg of memory which is ok for 1.5 on it but not the Sage which only has a half megabyte. I think I will port the 1.3 version as it will still run on both machines, and I hope the disk and most programs are still compatible between 1.3 and 1.5. That is next on my list as soon as the 68K EForth is on GEnie.

Which brings me to an end this time around. Check out GEnie and see some of the good Forth programs there. Join FIG. There is lots more support and other programmers willing to talk about how-to. I find Forth an excellent personal language. Keep programming and hacking.●

Z-Best Software

The Z3HELP System

By Bill Tishey

Z-System tools have been released at a steadily increasing rate over the past 3-4 years. Since the appearance of version 3.3, many programmers seem to have awakened to the power and potential of ZCPR and to new-found enjoyment in programming under CP/M. This has been great for Z users, who now have as wide a selection of system utilities as any UNIX user probably enjoys! At times, however, this flurry of development, has had one drawback—diminished attention to documentation. Updates to existing programs, in particular, have appeared so frequently that there's been an annoying lag between the release of a new version and appropriate documentation describing the new features and changes to previous usage. This seemed to be the case, at any rate, before ZSIG began to provide some control over the development process.

In response to the need for more up-to-date documentation, in late 1987, I began putting together a Help system for the Z utilities to keep track of the many new programs and new functions being added to older ones. I patterned the system after ALIASES.HLP, a 'Help System for Online Aliases' presented in Echelon's ZNEWS-letter #507. My goal was to make it both comprehensive (in terms of the programs which needed Help files) and practical (in terms of the information to be included in each file).

First, I had to decide what information about a utility I wanted to have 'on-line' in a Help (.HLP) file. I came up with the following categories:

1. A quick view of stats about the program: its size, crc, version number, date of issue, author, and where I could find it in my archived files.
2. A brief description of what function the program served.
3. A brief explanation of the program's syntax and any available options.
4. Special reminders concerning the program's operation, configuration, compatibility with other tools, et cetera.
5. Explanation of any error messages.
6. A history of changes from one version to another (usually as "Notes")
7. Examples of Use

I then set out to examine existing .HLP files, DOC files, the ZNEWS-letters, and even source code and BBS messages, to compile information in these categories for each utility. This was a monumental task, to say the least, and I soon decided first to develop a base of essential data on each utility and later to expand on such things as 'Notes' and 'Examples of Use' as time permitted.

Next, I wanted to be able to access this information quickly and easily. An alphabetical sorting by utility name, in a 'user-indexed' Help file, seemed logical, since this 1) would organize the information, 2) provide an easy menu with which to access it, and 3) provide an extensible system for future additions. The only question at this point was whether to provide a separate help file for each utility or to include all information in several large, user-indexed files.

While separate files might allow for easy updating, the idea of over a hundred .HLP files in a directory was rather distressing. Then came LBRHLP from Bob Peddicord—just what the doctor ordered. This super utility (now at version 1.8) allowed me to have separate files without the hassle of over-crowded directories. I simply created an 'A.HLP' file to serve as a menu, developed separate .HLP files for each utility beginning with 'A', and did a 'Group Build' on them with VLU to create a crunched 'A.LBR'. Defining 'D15:' as my 'default du:' for LBRHLP, I then set out to create a similar LBR for each letter of the alphabet to reside on D15:. [Note: The uncrunching routines in early versions of LBRHLP used up to 22k of buffer space, leaving only 25k+ for files. This resulted in 'memory overflow' messages when trying to read large help files such as DU3.HLP, VLU.HLP, VMENU.HLP, and help modules for some of the system libraries. With LBRHLP14, however, Howard Goldstein improved the buffer allocation, so that uncrunched files within a LBR could be an additional 24k in size.]

To simplify accessing the files, I created a series of aliases:

```
A LBRHLP -A A
B LBRHLP -B B
```

and a similar one for each letter of the

Bill Tishey has been a ZCPR user since 1985, when he found the right combination of ZCPR2 and Microsoft's Softcard CP/M for his three-year-old Apple II+. After graduating to ZCPR30 and PCPI's Applicard CP/M, he did a "manual install" of ZCPR3.3 (with help from a lot of friends!), and in late 1988 switched to NZCOM and ZSDOS, all on the same vintage Apple II+. Bill is the author of the Z3HELP system, a monthly-updated system of help files for Z-System programs, as well as comprehensive listings of available Z-System software. Bill is the editor of the Z-System Software Update Service and has compiled such offerings as the Z3COM package and the Z-System Programmer's Toolkit. Bill is a language analyst for the federal government and frequents the Foreign Language Forum (FLEFO) on Compuserve. He can be reached there (76320,22), on Genie (WATISHE), on Jay Sage's Z-Node #3 (617-965-7259) and by regular mail at 8335 Dubbs Drive, Severn, MD 21144.

Figure 1.

```
;
                    Help Menu for Z-System Utilities

Online utilities are described here.  To get help on  a command,  simply type
the corresponding letter.  To go back to CP/M, enter a ^C.

A - @        .COM                                                    - N
B - ABORT    .COM                                                    - O
C - AC       .COM                                                    - P
D - ACMDUTIL.COM                                                     - Q
E - ACOPY    .COM                                                    - R
F - ACREATE  .COM                                                    - S
G - ADIR     .COM                                                    - T
H - AFIND    .COM                                                    - U
I - ALIAS    .COM                                                    - V
J - ANY4     .Z80                                                    - W
K - ARRAYLIB.REL                                                     - X
L - ARUNZ    .COM                                                    - Y
M - ASK      .COM                                                    - Z
:a :AT
:b :ABORT
:c :AC
:d :ACMD
:e :ACOPY1
:f :ACREATE
:g :ADIR
:h :AFIND
:i :ALIAS1
:j :ANY4
:k :ARRAYLIB
:l :ARUNZ1
:m :ASK
```

alphabet and placed them in ALIAS.CMD in my ROOT directory. Then Jay Sage showed me how, instead of 26 such aliases, the following single alias could serve for all:

```
A=B=C=D=E=F=G=H=I=J=K=L=M=N=O=P=Q=R=..=W=X=Y=Z LBRHLP -$0 $0
```

...the $0 equating to whichever letter I pressed. [Ed: I have truncated the alias to fit on one line. Fill it out with each letter of the alphabet]. Now, all I had to do was type '<space>A' in any disk and user area, and 'A.HLP' would come up with a 'menu' of Help files for all utilities on the system beginning with 'A'. [Note: Z3PLUS users must use '/A' since '<space>A' had to be omitted as an option under Z3PLUS. '/A' will work for all Z-System users]. See *Figure 1* for A.HLP, the Z3HELP menu file for utilities beginning with "A".

Individual help files are accessed from such lettered menu files. To call ASK.HLP, for example, simply type "M" from the menu and LBRHLP will retrieve and uncrunch the appropriate file from A.LBR on D15:. The menu files can also be easily edited and expanded to suit a user's needs. When I get around to adding a Help file for Bridger Mitchell's command scheduler (AT05C.LBR), for example, I'll simply include "AT .COM - N" for option "N" in the menu section and add ":n :AT05C" in the information section to invoke AT05C.HLP for selection "N".

Figure 2 shows the header and initial screen for ASK.HLP, a typical user-indexed, "root" Help file from the Z3HELP system. The program name and "menu bar" are in reverse video which serves to neatly block off the program stats from the function description. The stats section may contain extra lines if Type-3 and Type-4 versions of a program are available. The menu bar will vary from one help file to another, but usually begins with "Syntax/Options" and ends with "Notes" and "Examples of Use". The function description is brief (often taken from the ".FOR" file provided in most distribution LBRs). Some utilities, of course, are released with already excellent Help files, and these are included in the system. Usually they are an additional, more detailed, source of information, accessed from the menu bar of the root Help file.

Updating the Help files has also proven to be simple. Using VLU to uncrunch/extract the .HLP file in question, I use ZDE to make appropriate changes, recrunch the file (supplying a new date of update), open the .LBR with NULU, and use option -R to replace the old .HZP with the new. This usually takes no longer than a few minutes. So, there you have it—a simple Help system, made 'simple' by the excellent tools (ARUNZ, VLU, LBRHLP, et cetera) already available for ZCPR3.

For the past several years I've been bundling individual updates of Help files into Z3HELPxx.LBRs and distributing them with the ZSUS monthly releases. The update LBRs are also available for download from Jay Sage's Z-Node #3 and from other popular Z-Nodes (Z3HELP40 is the latest as of this writing). A number of people have devised clever ways of automating the process of updating the Z3HELP system from the monthly Z3HELPxx.LBRs, and I'll describe a few of these in my next column. The entire Z3HELP system, comprised of five disks and current as of April 19, 1991, can be ordered through ZSUS for just $20 (a significant savings, no doubt, in what it would cost to download 1.5 megabytes of data!). For Z-Node sysops who would like to use the system on-line on their BBSs (presumably in a HELP directory), the price is but $10. As I've said before, error reports and suggestions for improving the system are always greatly appreciated.

Z-System LBR Tools

Bruce Morgen, Howard Goldstein, and Gene Pizzetta have been busy of late refining existing LBR utilities and even creating a few new ones. See *Figure* 3 for a list of the most

Figure 2.

```
                                ASK.COM

          Size (recs) CRC   Version    Author/Latest Issue      Disk
            6k (46)   1765  2.2        Richard Campbell 2/91    Z3COM1

   1- Syntax  2- Usage  3- Notes  4- Examples of Use


   ASK is a version of the MS-DOS utility which allows testing of user input
in batch files.  It is tailored for use with zex/alias/arunz scripts and will
only run on Z-System (ZCPR3x).  You must have the message buffers available
to make use of this program.  Written in BDS-C/Z vs 2.0 with routines from
Cam Cotrill's CFORZ02.
```

recent versions of available LBR tools.

Figure 3.

```
CL.COM       1.00   0 V2/09  4  31 FA0E LC10      25 05/07/91 Bruce Morgen
  Compacts a library, eliminating "dead space" caused by member files marked
  as "deleted". Can also erase members from the LBR, lists and wildcards
  allowed. For CP/M 2.2, with runtime ZCPR3 support.

èLBREXT.COM   3.20   0 V2/08  8  62 CE71 LBREXT32  35 04/06/91 Howard Goldstein
  Extracts crunched, squeezed and LZH-encoded files from LBRs. Vs 2.0 by Bob
  Peddicord.

LDIR-B.COM   1.80   0 V2/06  2  16 CD05 LDIRB18   30 02/17/91 Gene Pizzetta
  Displays library directories showing file dates and sizes. Includes summary
  line giving total member files found. Vs 1.0 by Steven Greenberg.

LGET.COM     1.10   0        4  30 2A8E LGET11     9 11/19/86 Bruce Morgen
  Extracts specified files from an indicated LBR. Vs 1.0 by Richard Conn.

LLF.COM      1.10   0        5  36 A1E1 LLF11     10 11/27/86 Bruce Morgen
  List Library files displays the directory of a declared library file. Vs
  1.0 by Richard Conn.

LPUT.COM     2.00   4 V208   6  48 8F34 LPUT20    36 04/06/91 Howard Goldstein
  Automated ZCPR3 library maker. Does for LBR creation what LGET does for
  extraction. Vs 1.0 by Bruce Morgen.

LREPAIR.COM  1.00   0 V209   2  16 8304 LREPAIR    9 05/05/91 Bruce Morgen
  LBR file integrity tester with limited correction capabilities. Matches
  CRCs of member files with values stored in LBR directory and fixes the CRC
  if the value is 0000h and the member file is not empty.
```

CL.COM

If you use LPUT to build a library (LBR) file, then decide to replace a member with a larger (or, for that matter, smaller) modified version, you've probably had to resort to NULU to deal with repacking the LBR. Until now. LPUT author Bruce Morgen has been putting the finishing touches to Compact Library, an idea originated several years ago by Michal Carson to compact a library in place and truncate it, leaving only the necessary blocks allocated. The program was dormant for a long time, since it was found to fail on some systems while writing the disk directory. Bruce took up the challenge to fix, improve and generalize it, however, and CL is now a welcome addition to the Z-System LBR toolset. See *Figure 4* for CL's syntax.

CL compresses an LBR, overwriting deleted entries and other unused sectors of the LBR file with active members. It then de-allocates the remaining blocks and/or extents of the LBR file. CL will 1) compact an LBR in place, 2) report the amount of free space remaining in an LBR, 3) force a compaction, or 4) delete members from an LBR. Compacting is not done if none is required, even if the "Z" option is selected. Directory sorting is done any time there is a compacting operation, whether forced via the "Z" option or decided on by CL itself. Only the sort is done if there's no need for compacting and the "Z" option is selected. See *Figure 5* for some examples of CL's usage.

Giving CL the ability to delete a library member was a stroke of genius and makes it the perfect "Pack/Erase" utility to complement existing LBR tools. [Note: CL may not be compatible with all systems, so caution is advised. CL does not create a new library. It overwrites the existing library. Also, CL cannot currently Krunch a file (ala NULU) to make a larger directory. An enhanced LPUT might well be better equipped to do this than CL, since LPUT already has the code to build a new LBR and to add members.]

LREPAIR.COM

Bruce Morgen's LREPAIR is a new LBR utility which checks CRCs and repairs LBRs built by old or non-LUDEF-compliant programs. LREPAIR is loosely derived from Sigi Kluger's LCRCK v1.10. It tests LBR file integrity by checking member file CRCs against LBR directory data. It will correct member CRCs when they are invalid 0000h values as created by LBRDISK and early versions of LU and NULU, and will also fix the directory CRC in such cases. Uncorrected member CRC errors are reported to the Program Error Flag (a value of 255 indicates the LBR directory is probably corrupted, 254 tells you that no members files were found). Another instance in which LREPAIR corrects the LBR directory CRC is when it strips any high bits encountered in member files' names. See *Figure 6* for LREPAIR's syntax and an example of usage.

Other LBR Tools

Bob Peddicord's LBREXT32, Bruce's LPUT20, and Steven Greenberg's LDIRB18 are undoubtedly the best trio of tools for extracting, adding, and listing member files of LBRs. The latest versions of LBREXT and LPUT (courtesy of Howard Goldstein) store and retrieve the create and modify times as well as dates from the LBR's directory. LPUT also now allows an input filespec or dir: which is equivalent to dir*.*. The latest enhancement to LDIRB by Gene Pizzetta is a count of the member files to be displayed, a nice addition to such LDIRB niceties as display of member file creation and modification dates, as well as the original names of squeezed, crunched, and "LZHed" members.

We should not overlook ZCPR 3.0 author Richard Conn's LLF, however, as an alternative to LDIRB. While LDIRB is a more informative LBR lister with regard to member file history, LLF can show member file CRCs (which LDIRB at present does not) and member file indices (which only the larger LBR managers, NULU and LU, can do). Aware of these advantages, Bruce has provided an update to LLF (LLF12PAT.LBR) which fixes a cosmetic flaw on abort.

Bruce has also provided a patch (LFINDPAT.LBR) to give Z-System support to Martin Murray's LFIND, which searches one or more LBRs for a specified file. A patched LFIND 1.13z will extract the maximum drive and user from the environment if running under Z33 or above and will respond to "//" help queries.

Figure 4.

```
CL (Compact Library), Version 1.0
Syntax:
   CL [DU:]filename[.LBR]                    compact filename
   CL [DU:]filename[.LBR] ?                  report free space
   CL [DU:]filename[.LBR] Z                  force compact
   CL [DU:]filename[.LBR] -afn1[,afn2,...]   delete members
```

Figure 5.

NULU is used to delete one of 3 member files from SCATV208.LBR. CL is run (Step 1) to show that 25 records of free space (4k) exist, represented by the deleted member. CL is run again (Step 2) to compact the library and reclaim the 4k. The far right column (20k) indicates the resulting size of the LBR. CL is run again (Step 3) and shows that no free space remains to be claimed. The deleted member file is replaced using LPUT, and CL is used this time (Step 4) to delete it. CL is then run with the "Z" option (Step 5) to force a compaction. Since a compaction is not required, the directory is simply sorted and checked.

```
1) B0:WORK>cl d3:scatv208 ?

CL (Compact Library), Version 1.0
25 records of free space in D3:SCATV208.LBR      24K          not compacted
Contents: 2 active members, 1 deleted member, 4 open member slots.

2) B0:WORK>cl d3:scatv208

CL (Compact Library), Version 1.0
25 records of free space in D3:SCATV208.LBR      24K          20K
Contents: 2 active members, no deleted members, 5 open member slots.

3) B0:WORK>cl d3:scatv208 ?

CL (Compact Library), Version 1.0
0 records of free space in D3:SCATV208.LBR       20K          not compacted
Contents: 2 active members, no deleted members, 5 open member slots.

4) B0:WORK>cl d3:scatv208 -zsusv208.czt

èCL (Compact Library), Version 1.0
ZSUSV208.CZT deleted.
25 records of free space in D3:SCATV208.LBR      24K          20K
Contents: 2 active members, no deleted members, 5 open member slots.

5) B0:WORK>cl d3:scatv208 z

CL (Compact Library), Version 1.0
0 records of free space in D3:SCATV208.LBR       20K          sorted & checked
```

Z Message Base

In gathering the updates and new program releases for ZSUS from the many RASes supporting Z-System, I also take time to read many of the messages between developers, programmers, and users. Since I capture, edit and organize many of these messages, anyway, it seemed like a good idea to share a select few with you each time at the end of this column.

Z-Node #3, 04/17/91, TYPE 3 AND TYPE 4 PROGRAMS

Question: Although I have been running NZCOM for some two years now, up to this point I have completely ignored using Type 3 and Type 4 versions of programs simply because I don't understand this concept. I realize that these versions are intended to run faster, but I don't know enough about their limitations, if any. Are they preferable for anyone running NZCOM? Is there the danger that they will somehow get in the way of other programs? How do you decide between choosing a Type 3 or Type 4 program? I suspect I'm not the only one in the dark in this matter, and I'd appreciate some guidance about using these versions.

Answer: A type-3 program is just like a type-1 (standard program) except that it loads and runs at an arbitrary address, while standard programs run only at 100H. There is no speed difference at all. The advantage is that a type-3 program will generally not clobber the application program that you had in memory. Thus they are particularly suitable for programs that the system runs automatically, sometimes without the user's knowledge even, such as error handlers and extended command processors and shells. Type-4 programs are functionally the same as type-3 programs except that they figure out where to load themselves at the time they load. All the ones released to date automatically load themselves as high in memory as possible. Because of the address relocation required, load time is very slightly longer (probably imperceptibly); also the file is a little longer because of the header and relocation map. The type-4 program is either less likely to clobber a loaded application (because it loads higher) or will work when type-3s will not (if there is not enough high memory for them). Thus, type-4s are generally to be preferred. Perhaps I should have said earlier that all this matters only if you use the GO command. Also, my Z33 User Guide discusses type-3 programs in detail! (Jay Sage, Sysop)

Z-Node #77, 4/30/91, NZCOM-DBASE II WARNING.....

I just reviewed Bruce Morgen's suggestion for solving the problem of the placement of the $$$.SUB file when using dBase-II with ZCPR 3.3 and ZCPR 3.4. This suggestion came packaged with a recent release of NZCOM, and suggests that the user use a small binary file with the dBase CALL command which will switch to user 0 prior to exiting dBase with the QUIT TO command. Bruce correctly points out that switching to user 0 will cause dBase to write the $$$.SUB file to a0:, which is where ZCPR3.3 and 3.4 look for it. THIS METHOD SHOULD BE USED WITH >>EXTREME CAUTION<<. dBase-II maintains internal buffers containing database information and does a certain amount of housekeeping—including disk writes—on exit. If the user area has been changed, databases can be lost or corrupted. IF YOU USE THIS PROCEDURE, BE SURE TO EXPLICITLY CLOSE ALL OPEN FILES PRIOR TO CHANGING USER AREAS. The USE command, with no arguments, applied to both the primary and (if any) secondary databases will accomplish this, and I would advise using the CLEAR command as well. (Lindsay Haisley, Sysop)●

Figure 6.

```
B0:WORK>lrepair //

LREPAIR, Version 1.0
Checks CRC of all or selected member files in
an LBR, corrects invalid 0000h CRC values, and
strips high bits from member file names.
Syntax:
 LREPAIR [dir:]lbrname[.LBR] [afn.typ]

B0:WORK>lrepair multiprt

LREPAIR, Version 1.0
Checking B0:MULTIPRT.LBR
CRC for PR       .ALI is 8392h  (OK)
CRC for PRINT    .ZEX is DE66h  (OK)
```

Stepped Inference as a Technique

for Intelligent Real-time Embedded Control

By Matt Mercaldo

In the last article of this series I promised feedback for our system of motors. Well, Its been a very short bi-month (I couldn't get my hands on an encoder to complete this project). Instead of giving you feedback, this article will concern itself with the Stepped Inference model touched on in the last article. With the advance of embedded processors and the need for concise and standard expressions of real-time state oriented models, Stepped Inference offers a mechanism for the concise expression of complex state oriented models. This article will discuss the stepped inference model, the internals of the inferencing mechanism, and some application of the stepped inference model.

Real Time Control

Typically real time control concerns itself with the reception, analysis and reaction to a "real world" environment. The real time system has a certain perspective on the reality it has been focused upon through its hardware and software architecture. At predicted times, certain events will trigger certain of the system's actions. An event can be thought of as any single monitored occurrence or combination of monitored occurrences. Actions change both the real and or perceived reality. From the real-time system's perspective there are two realities: the real and the perceived. The "real" reality is that reality which directly acts upon the real time system. The perceived reality is that perspective or aspect of the real reality which is accounted for within the system and is used indirectly for the control or manipulation of the real reality. Data objects and their associated methods typically embody the "consciousness" of perceived reality.

Real time systems are usually modeled with state transition diagrams, data dictionaries, and transformation specifications. From these techniques an important framework of data objects and functional relationships are developed. These can be used to directly define the system in terms of stepped inference.

The Stepped Inference Model

Stepped inference is a technique whereby states of a multi-state model are expressed in rule sets, and upon any given expected event the current rule set will dictate action. Events can be any combination of expected external or internal stimulus. An inferencing process is induced by any received event. Perception of the event occurs when the correct rule is resolved. Associated with each rule is an action that alters either the perceived reality or the real reality or both.

Stepped inference is a reflexive system that dictates action based on events. These events can originate from any source, internal to the processor or external to the processor. Actions can be direct or indirect. Direct actions alter reality while indirect actions alter data objects representing the perceived reality.

The knowledge of how to handle specific states is kept within rule sets. A rule is an object that contains a pointer to a condition function followed by a pointer to an action function. The condition function determines if a certain perception has occurred (A perception being a group of events). Instance information is not embedded within any given rule. The instance information is specific to activator context (activator being that which effects action). Instance information is contained within a control block specific to the activator. Rules should only know how to control a class of activator, not a specific instance of activator. In Article III of this series, rules and rule sets have the knowledge of how to run a stepper motor while the motor control block (MCB) has the necessary instance information for a specific motor.

When an event occurs, an inferencing process is initiated. The inferencing process is done with a tightly coded inference engine that does a partial inference of a context of rules (group of rule sets). The engine runs through the current rule set attempting to synchronize on an event. One of the rules in the rule set will resolve and fire its action. If the condition portion of the current rule returns true then the action portion of the rule is fired and the inference cycle is complete. If the condition portion of the current rule returns false then the next rule is made current and the inferencing mechanism calls itself recursively. One of the rules in the rule set must fire. Rule priority is based on a rule's position in the rule set. The closer to the top of the rule set, the higher the priority.

The inference engine is typically run at interrupt level. For example, a regular clock interrupt can invoke the inferencing process. This is how the motor control example runs. Care must be exercised if the inferencing mechanism is run at interrupt level. The inference latency (time to complete an inference

See Stepped Inference, page 54

Matthew Mercaldo is employed by a huge firm. With a small group, he develops software tools for field service engineers to do their thing. At 4:30 or 5:00 p.m., when the whistle blows, his thoughts race toward the edge. He dreams of articulated six legged walking beasts, electronic brains that can fend for themselves, and the stuff of "U.S. Robots and Mechanical Men." Someday he dreams of running power out to his garage, and with his wife and a select group of friends, opening his own automoton shop - and thus partially fulfilling his childhood dreams. (Plutonium, Tritium and the like are still not available for public "consumption"; but seeing the moons of Jupiter would be spectacular in one's own starcruiser!)

```
;
-gethi: djnz    r11,-echo
        ld      r12,r14         ; get high adrs byte in data reg
        jr      -echo1          ; and go send it

;
; 29H: read back contents of data register
;
-echo:  djnz    r11,-fetch
-echo1: txon                    ; turn on transmitter (tx buf is empty)
        ld      r11,r12         ; transmit high nybble
        swap    r11
        and     r11,#0fh
        or      r11,#30h
        and     irq,#0efh ; clear the irq bit
        ld      sio,r11
-tx1:   tm      irq,#10h        ; wait for tx buf to empty
        jr      z,-tx1

        ld      r11,r12         ; transmit low nybble
        and     r11,#0fh
        or      r11,#30h
        and     irq,#0efh ; clear the irq bit
        ld      sio,r11
-tx2:   tm      irq,#10h        ; wait for tx buf to empty
        jr      z,-tx2
        txoff                   ; turn off transmitter
        ret

;
; 2AH: fetch byte from memory or register, and transmit
;
-fetch: djnz    r11,-store
        ; the fetch operations can go one of five ways, depending on r13
        ld      r11,r13
-efetch: djnz   r11,-ifetch             ; 1: E memory
         lde    r12,@rr14
         jr     -fdone
-ifetch: djnz   r11,-cfetch             ; 2: indirect reg
         ld     r12,@r15
         jr     -fdone
-cfetch: ldc    r12,@rr14               ; 0: C memory
-fdone: incw    rr14
        ; now output two hex digits from the data register just fetched
        jr      -echo1

;
; 2BH: store byte in memory or register, per bank select
;
-store: djnz    r11,-setlo
        ; the store operations can go one of five ways, depending on r13
        ld      r11,r13
-estore: djnz   r11,-istore             ; 1: E memory
         lde    @rr14,r12
         jr     -sdone
-istore: djnz   r11,-cstore             ; 2: indirect reg
         ld     @r15,r12
         jr     -sdone
-cstore: ldc    @rr14,r12               ; 0: C memory
-sdone: incw    rr14
        ret

;
; 2CH: set low address byte
;
-setlo: djnz    r11,-sethi
        ld      r15,r12
        ret

;
; 2DH: set high address byte
;
-sethi: djnz    r11,-setext
        ld      r14,r12
        ret

;
; 2EH: set extended address byte (memory page)
;  Note: out of range values will default to C memory, later
;
```

Z8 Talker, from page 20

all set the memory bank register in the target.

With these words, it is straightforward to implement words such as DUMP and FILL.

Breakpoints

The breakpoint words allow up to ten breakpoints to be set in the target. When a breakpoint is set, its address and the three bytes replaced by the CALL must be saved. These are kept in BPARRAY. When a breakpoint is not in use, the address in BPARRAY is set to zero. (This means you can't set breakpoints at location zero...which is just as well, since those are interrupt vectors.)

INITBP simply marks all of the breakpoints as "unused".

BP0 through BP9 are the words which set breakpoints. They use the common word BREAK. If an existing breakpoint is being changed, the old three-byte code fragment must be restored first; this is done by -BREAK ("un-break"). Then !BREAK fetches the three bytes at the new location, and sets a breakpoint there.

When breakpoints are set, the application program should be started with GO. This uses AWAIT to listen for the "*" sent when a breakpoint is encountered; ?BREAK then fetches the flags and breakpoint address from the target, deduces which of the ten breakpoints was encountered, and displays a message.

HEX File Load and Save

LOAD loads an Intel hex file into the target system; SAVE saves a given range of target memory in an Intel hex file. These words are rather cryptic, but (I hope) easy to figure out.

These are the only words in the host program which do file access; they will need to be rewritten for your Forth system.

Using the Talker

1. Your Z8 board must be running the talker program.

2. The Z8's serial port must be connected to the PC's COM1 port. This may be either a full-duplex connection (separate wires for TXD and RXD), or a half-duplex RS-485 connection. For the half-duplex connection you should use a 75176 or DS3695 transceiver, with direction controlled by the COM1 port's DTR line.

3. The IBM PC must run the talker

host program, ZTALK.FTH.

The command set, in *Figure 3*, has been loosely modelled on the Zilog Super8 debug monitor. All numbers are entered and displayed in hex. You may put more than one command on a line.

If the talker program fails to respond while the host is waiting for data, the host will halt. This can happen if the target is dead and you attempt to do a DUMP or SET command—the host will send commands to a dead target, and then wait for a reply. Use ESC to escape this wait loop.

If this happens in the midst of a dump, or at random intervals, check that your serial line is clean. If you have embedded the TALK routine in an application, you may be polling TALK too infrequently. Type SLOW in the host and see if the problem goes away.

I almost always start a debugging session by entering TERM and then typing a couple of "*" characters. This is the memory examine command. If the talker is running, you will see a pair of characters each time you press "*".

Eventually I plan to improve the host program by using the command 29 hex to verify that data appears correctly in the MDR register. This would allow the host to automatically test for a "live" target, and to verify the serial line. But that's for another day...

Conclusion

I hope that you find this program, or a variant, useful in your work. I have attempted to generalize the functions of this talker so that they can be used with any 8-bit CPU. The program can easily be extended for 16-bit CPUs and larger address spaces. I'd be interested to hear of any of these.

The Z8 and Forth source code can be found in the Forth Interest Group Roundtable on GEnie, in file Z8TALKER.ZIP.

Credit for the technique of shifting data into MDR and MAR belongs to Dr. Don Schertz of Bradley University, who used it many years ago in an 8080 monitor program. I had the good fortune to be one of his students at the time.●

> "You have not converted a man because you have silenced him."
>
> —Lord Morley

```
-setext: djnz   r11,-go
         ld     r13,r12
-nofunc: ret

;
; 2FH: go to given address (resume execution at given address)
;
-go:    djnz    r11,-nofunc
        incw    sph             ; drop return adrs in 'talker'
        incw    sph
        ld      flags,r12       ; restore saved flags (if any)
        ei
        jp      @mar            ; go to address in monitor adrs reg (rr14)
```

```
\ FIGURE 2. THE HOST PROGRAM

\ SUPER8 "MICRO-TALKER" host program for IBM PC
\   (c) 1989,1990 T-Recursive Technology
\   placed into the public domain for free and unrestricted use
\
\ This is the host program for the minimal serial monitor
\ program, with customization for the Zilog Super8.
\
\ vers 1.0  original program under real-Forth for the IBM PC
\
\ vers 1.1  26 Jun 90                   bjr
\           modified talker function codes; added breakpoint
\           functions; various minor improvements
\
\ vers 1.2  2 Jul 90                    bjr
\           converted from screen file to text file; modified
\           for L.O.V.E.-83Forth; added multiple breakpoints;
\           various minor improvements
\
\ vers 1.3  10 Oct 90                   bjr
\           converted for 6-port SIO card using 2681, (2) Z8530
\
\ vers 1.4  17 Dec 90                   bjr
\           modified for MPE PowerForth; added support for
\           COM1 thru COM4
\
\ vers 1.4TCJ  26 May 91                bjr
\           removed support for 6-port SIO card and COM2:-COM4:;
\           removed Super8-specific words, for TCJ article
\
VOCABULARY TALKER  ONLY FORTH ALSO TALKER DEFINITIONS

\ ******************** BASIC SERIAL I/O ************************
HEX

\ port select parameters
VARIABLE 'STSREG        \ status register, for multiport words
VARIABLE 'DTAREG        \ data register, for multiport words

3F8 CONSTANT COM1PORT  \ COM1: base address (data reg, sts is at 5 + )
2F8 CONSTANT COM2PORT  \ COM2:
3E8 CONSTANT COM3PORT  \ COM3:
2E8 CONSTANT COM4PORT  \ COM4:

: COM1:   COM1PORT 5 + 'STSREG !  COM1PORT 'DTAREG !  ;

\ ?TX  returns true if transmitter is ready for a character
: ?TX   'STSREG @ PC@ 20 AND ;       \ COM ports

\ ?RX  returns true if receiver has a character
: ?RX   'STSREG @ PC@ 1 AND ;        \ DUART, SCC, and COM ports

\ TXON  uses the DTR line to switch a remote 75176 RS-485
\  transceiver to the "transmit" mode.
: TXON    0 'DTAREG @ 4 + PC! ;                  \ COM ports

\ TXOFF  switches the remote transceiver to the "receive" mode.
\  It waits for the last character to clear the tx.
: TXOFF    BEGIN  'STSREG @ PC@  40 AND UNTIL  3 'DTAREG @ 4 + PC! ;

\ (TX)  transmits a character thru the SIO port.  Note that it
\  DOES NOT wait for the UART to be ready.
: (TX)   'DTAREG @ PC! ;     \ all devices

\ (RX)  gets a character from the SIO port.  Note that it
```

```
\  DOES NOT wait for the UART to be ready.
: (RX)  'DTAREG @ PC@ ;    \ all devices

\ PACE is used to slow the PC down for slow target CPUs
VARIABLE PACE  DECIMAL  \ approx 1.5 usec per count w/ 9.54 MHz V30
: FAST        1 PACE ! ;
: MEDIUM    500 PACE ! ;
: SLOW    10000 PACE ! ;
: PACED   PACE @ 0 DO LOOP ;
HEX

\ ***************  UART INITIALIZATION  ********************

CREATE BAUDTABLE  ( COMn:, 4800 baud, N-8-1 )
   6 C,  \ 6 pairs follow
   3F8 1 + ,  0 C,   \ disable uart irpts
   3F8 3 + ,  80 C,  \ enable divisor latch
   3F8 ,      18 C,  \ divisor low
   3F8 1 + ,  0 C,   \ divisor high
   3F8 3 + ,  3 C,   \ 8 bits, no parity, 1 stop
   3F8 4 + ,  3 C,   \ modem ctl: RTS, DTR

\ General initialization program...expects a count, followed
\ by a list of  port # (low byte of port adrs), data  pairs
: S-INIT ( a )
   BAUDTABLE   DUP C@ 0 ?DO
      1+ DUP 2+ DUP  C@ ( data ) ROT @ ( port ) PC!
   LOOP DROP ;

\ ************** QUICK & DIRTY TERMINAL PROGRAM ****************
HEX
: COLOR   CURR-ATTRIBS ! ;     \ change screen color in PForth
07 constant white
70 constant yellow   \ actually this is black on white

\ We redefine TX to use the half-duplex serial link.  It turns
\ the transmitter on before sending, sends, then waits for the
\ character to finish and turns the transmitter off.
\ If using a full duplex link, this definition can be omitted.
: TRAP      KEY? IF ." *escape*" KEY DROP QUIT THEN ;
: TX ( c )   PACED  BEGIN TRAP ?TX UNTIL  TXON (TX) TXOFF ;
: RX ( - c )  BEGIN TRAP ?RX UNTIL  (RX) ;

\ TERM  is a simple terminal program.  ESC exits back to Forth.
: (TERM)   BEGIN   ?RX IF  (RX) EMIT  THEN
      ?TX IF  KEY? IF  KEY DUP
         1B = IF  DROP EXIT  ELSE  TXON (TX) TXOFF  THEN
      THEN THEN
   AGAIN ;

: TERM   YELLOW COLOR  (TERM)  WHITE COLOR ;

\ ***************** SUPPORT FUNCTIONS FOR TALKER ****************
HEX
\ SPILL  will empty the UART receiver (read and discard chars
\  until the UART is empty).  This is frequently necessary,
\  e.g., when external serial switches are used, or if the
\  target malfunctions.  ANY garbage characters in the rx data
\  stream will "unsynchronize" the talker and cause permanent
\  confusion, so it's good to SPILL at frequent intervals.
: SPILL   BEGIN ?RX WHILE RX DROP REPEAT ;

\ NYBLIZE  takes an 8-bit value, and converts it to two
\  pseudo-hex characters (ASCII characters from 30 to 3F hex).
: NYBLIZE ( c - lo hi )  0FF AND 10 /MOD  SWAP 30 + SWAP 30 + ;

\ DENYBL  takes two pseudo-hex characters (30 to 3F hex), and
\  converts them to a byte value (00 to FF).
: DENYBL  ( hi lo - c )   0F AND SWAP 0F AND 10 * OR ;

\ ><  swaps the hi and lo bytes of the top stack item.
CREATE SWAPPER  3 ALLOT
: >< ( n - n )   SWAPPER !  SWAPPER C@ SWAPPER 2+ C!  SWAPPER 1+ @ ;

\ >BYTES  splits the top stack item into its high and low bytes
\ >WORD  takes high and low byte values and merges them into
\  a 16-bit word value.
: >BYTES ( n - lo hi )   DUP >< ;   ( note hi 8 bits unknown )
: >WORD ( hi lo - n )   SWAP >< OR ;   ( hi 8 bits must be 0 )

\ TXH  transmits a byte as two pseudo-hex characters.
```

Z Corner, page 36

output facilities (such as keyboard redefiners or redirection of screen or printer output to disk). Until you try NZCOM or Z3PLUS, it is hard to imagine how easy it is to do these things.

Sources of Support

Here are three magazines that provide significant CP/M support. By reading them you will learn about sources for public-domain software (both by modem and on disk) and about vendors who sell CP/M commercial software (a vendor listing, CPMSRC.LST, posted on bulletin boards runs to 15 printed pages).

The major, professional publication with strong coverage of Z-System and CP/M is *The Computer Journal*. Its six issues per year have a broad hobbyist focus, with articles on several operating systems, programming languages, embedded controllers, hardware projects, and many other topics.

The Computer Journal ($18/yr)
P.O. Box 12
S. Plainfield, NJ 07080-0012
voice: 908-755-6186
modem (Z-Node #32): 908-754-9067
on GEnie as TCJ$

Two other, more informal magazines of interest to the 8-bit community are:

The Z-Letter ($15/yr)
Lambda Software Publishing
720 S. Second Street
San Jose, CA 95112
voice: 408-293-5176

Eight Bits and Change ($15/yr)
Small Computer Support
24 East Cedar Street
Newington, CT 06111●

Sponsor a Friend

A reminder to subscribers: when a friend enters a prepaid subscription as a result of your discussions, you are entitled to an extension of one issue to your own subscription for free. Be sure your friend identifies you as the sponsor when ordering.

```
\ RXH  receives a byte as two pseudo-hex characters.
: TXH ( n )   NYBLIZE TX TX ;    ( transmit a hex value )
: RXH ( - n )   RX RX DENYBL ;   ( receive a hex value )

\ ************* PRIMITIVE TALKER OPERATIONS **************
HEX
\ XADR  sends the "current address" to the target.
\  This is done as: send hi byte, send "hi adrs" command,
\  send lo byte, send "lo adrs" command.
\  We do a SPILL as part of XADR because XADR is used by
\  everything, and when it is used we aren't expecting data.
: XADR ( a ) SPILL >BYTES TXH 2D TX ( hi ) TXH 2C TX ( lo ) ;

\ MPGE builds words which send a page-select command to the
\  target. Pages are selected by an 8-bit "page address."
\ CMEM, EMEM, and REGS are the memory pages defined
\  for the Z8 talker program.
: MPGE   CREATE  C,  DOES> C@ TXH  2E TX ;
0 MPGE CMEM     \ "C" code memory
1 MPGE EMEM     \ "E" external memory
2 MPGE REGS     \ registers

\ X@+  fetches a byte from the target at the current address.
\ X!+  stores a byte in the target at the current address.
\  Both of these functions use the currently selected memory
\  page, and increment the current address when done.
: X@+ ( - n )   2A TX  RXH ;
: X!+ ( n )   TXH  2B TX ;

\ JUMP  starts target execution at the current address.
\  The Z8 talker expects the flag byte to be in the talker
\  data register when a "go" is issued. This words sends a zero
\  flag byte, which will force bank 0 (also no fast irpt).
: JUMP ( a )   XADR  0 TXH 2F TX ;

\ ************ MEMORY/REGISTER DUMP AND EDIT *************
HEX
\ ?EMIT  filters nonprintable characters.
: ?EMIT ( c )  7F AND DUP 20 < IF DROP ASCII . THEN EMIT ;

\ DUMP  dumps a range of memory or registers.
: DUMP ( a n )   OVER + SWAP DO   CR I 5 U.R  2 SPACES
      I XADR  I 10 OVER + SWAP DO  X@+ 3 .R LOOP  2 SPACES
      I XADR  I 10 OVER + SWAP DO  X@+ ?EMIT  LOOP
   10 +LOOP ;

\ FILL  fills a range of memory with a byte value c.
: FILL ( a n c )  ROT XADR  SWAP 0 DO  DUP X!+  LOOP  DROP ;

\ SET  is an interactive examine/alter of memory or registers,
\  modelled after the Zilog Super8 monitor "set" command.
\  Given an address, it displays its contents. The operator
\  may then type <CR> to advance to the next address, or a
\  Forth expression followed by <CR> to store a new value and
\  advance.  The loop is exited with Q <CR> .
: SET ( a )   BEGIN  CR DUP 5 U.R  DUP XADR X@+ 3 .R  SPACE
   DEPTH >R QUERY INTERPRET   ( parse a line of Forth )
   DEPTH R> - 1 - 0= IF  ( if depth is +1, ... adr n on stack )
      OVER XADR  X!+
   THEN 1+ AGAIN ;

: Q ( a )  DROP QUIT ;  ( drop the adrs & return to Forth )

\ ***************** BREAKPOINT FUNCTIONS *****************

\ BPARRAY  is used to "remember" up to 10 breakpoints. For each
\  we must remember the 2-byte address of the breakpoint, and
\  the 3 bytes at that address which were replaced by the
\  breakpoint CALL instruction. We allot 3 cells (6 bytes).
\ BPADR  returns the "address storage" location for breakpt n.
\  A breakpoint is flagged 'inactive' by storing an address of
\  zero. (So, you can't set breakpoints at address zero.)
\ BPMEM returns the "target memory storage" location for b.p. n
CREATE BPARRAY   DECIMAL 10 6 * ALLOT
: BPADR ( n - adr )   6 * BPARRAY + ;
: BPMEM ( n - adr )   6 * BPARRAY + 2 + ;

\ INITBP initializes all breakpoints (clears the breakpoints
\  without attempting to restore memory contents).
: INITBP   10 0 DO  0 I BPADR !  LOOP ;
```

```
INITBP

\ .BP  prints the addresses of all the currently active
\  (non-zero) breakpoints.
: .BP   CR ." Breakpoints are set at:"  10 0 DO
      I BPADR @  ?DUP IF  CR I 3 .R  7 U.R  THEN
   LOOP ;

\ FINDBP  finds the breakpoint with the given address.
\  If none found, -1 is returned.
: FINDBP ( adr - n )   -1  10 0 DO
      OVER I BPADR @ =  IF DROP I LEAVE THEN
   LOOP SWAP DROP ;

HEX
\ !BREAK  sets breakpoint n at the given address.
\  Note that this selects CMEM.
: !BREAK ( adr n )   2DUP BPADR !  ( save the address )
   CMEM OVER XADR  X@+ X@+ X@+   ( get old contents )
   >WORD  ROT BPMEM 2!           ( save old contents )
   XADR 26 TX ;          ( send adrs, request breakpoint )

\ -BREAK  restores the previous contents of breakpoint n,
\  and marks breakpoint n as inactive.
\  If breakpoint n was not set, no action is taken.
\  Note that this may select CMEM.
: -BREAK ( n )   DUP BPADR @ ?DUP IF   ( only if active! )
      CMEM XADR          ( set the address in the target )
      DUP BPMEM 2@ >BYTES ROT  ( get old contents, reversed )
      X!+ X!+ X!+           ( restore old contents )
      0 OVER BPADR !        ( mark this b.p. cleared )
   THEN DROP ;

\ BREAK sets breakpoint 'n' at the given address.
\  The previous breakpoint 'n' (0-9) is cleared.
\  If the new breakpoint address is zero, this
\  simply clears breakpoint 'n'.
: BREAK ( adr n ) DUP -BREAK  OVER IF !BREAK ELSE 2DROP THEN ;

\ BREAKPOINT  is used to build BP0 thru BP9.
\ BP0 thru BP9  are simply shorthand versions of
\  "0 BREAK" thru "9 BREAK", respectively.
: BREAKPOINT ( n )   CREATE C, DOES> C@ BREAK ;
0 BREAKPOINT BP0  1 BREAKPOINT BP1  2 BREAKPOINT BP2
3 BREAKPOINT BP3  4 BREAKPOINT BP4  5 BREAKPOINT BP5
6 BREAKPOINT BP6  7 BREAKPOINT BP7  8 BREAKPOINT BP8
9 BREAKPOINT BP9

\ ?BREAK gets the target program status upon occurence of a
\  breakpoint.  The Z8, on a break, leaves the return
\  address in the talker address register, and the CPU flags
\  in the talker data register. Note that the flags must be
\  read first, since all other operations destroy the data reg.
\  Note also that the breakpoint address is 3 less than the
\  return address (a Z8 CALL is 3 bytes).
\  The address and flags are saved in variables for RESUME.
VARIABLE RESUMEADR   VARIABLE RESUMEFLAG
: ?BREAK   29 TX RXH ( flags )  RESUMEFLAG !
   28 TX RXH ( ahi ) 27 TX RXH ( alo ) >WORD 3 - RESUMEADR !
   CR ." * BREAKPOINT " RESUMEADR @ FINDBP
   DUP 0< IF  ." unknown" DROP  ELSE  1 .R  THEN
   ."  at " RESUMEADR @ U.
   ."  flags=" RESUMEFLAG @ 2 .R ;

\ AWAIT waits for a character to be received from the target.
\  If it is an '*', get and print the breakpoint info.
\  The talker sends an 'M' when the monitor is initialized,
\  an '*' when breakpoint encountered.
: AWAIT   RX ASCII * = IF ?BREAK THEN ;

\ GO  starts execution at a given address, and AWAITs.
: GO ( adr )   JUMP AWAIT ;

\ RESUME continues execution after a breakpoint, and AWAITs.
\  Execution continues at the address of the breakpoint call.
\  NOTE! if you don't reset the breakpoint, you simply
\  trip it again instantly!
: RESUME  RESUMEADR @ XADR RESUMEFLAG @ TXH 2F TX AWAIT ;

\ *************** HEX FILE LOAD AND SAVE ***************
HEX
```

```
PCB HEXFILE
VARIABLE CSUM          \ used for checksum calculation
VARIABLE HEXCHAR       \ can only READ and WRITE from memory!

\ hex-to-ASCII conversion table
CREATE HEXASC 30 C, 31 C, 32 C, 33 C, 34 C, 35 C, 36 C, 37 C,
     38 C, 39 C, 41 C, 42 C, 43 C, 44 C, 45 C, 46 C,

\ PUT  writes a character to the currently open file
\ PUTH  puts a single hex digit
\ PUTHH  puts a byte value as 2 hex digits, hi & lo
: PUT ( c ) HEXCHAR C! HEXCHAR 1 HEXFILE HANDLE WRITE-PATH
   ABORT" Error writing file." DROP ;
: PUTH ( n ) HEXASC + C@ PUT ;
: PUTHH ( n ) DUP CSUM +! DUP 0F0 AND 10 / PUTH 0F AND PUTH ;

\ WREC  writes a single record (line) of an Intel hex file,
\  starting at memory address a, for n bytes. The record type
\  is typ. If 0 bytes are requested, only the header info is
\  written.
: WREC ( typ a n )   0 CSUM !  ASCII : PUT ( rec start )
   DUP PUTHH ( length ) OVER >< PUTHH OVER PUTHH ( adrs hi,lo )
   ROT PUTHH ( type )   DUP IF ( length>0 )
      OVER 0D EMIT 4 U.R
      SWAP XADR  0 DO  X@+ PUTHH  LOOP     ( data bytes )
   ELSE 2DROP THEN
   CSUM @ NEGATE PUTHH ( checksum )  0D PUT 0A PUT ( cr,lf ) ;

\ SAVE given an address and length, writes it as a hex file.
\  The hex records will be no more than 16 data bytes long.
\  Note: maximum SAVE length is 32K.
: SAVE ( a n ) CMEM HEXFILE PATHNAME "" .HEX" HEXFILE SET-EXT
   HEXFILE CREATE-PATH-PCB
   ABORT" Can't create file."  CR
   BEGIN  DUP 0>  WHILE
      2DUP 10 MIN  0 ROT ROT  WREC  ( write <= 16 bytes )
      SWAP 10 + SWAP 10 - REPEAT     ( +address, -length )
   2DROP  1 0 0 WREC
   HEXFILE HANDLE CLOSE-PATH ABORT" Can't close file." ;

\ GET  reads a character from the currently open file
\ GET:  scans the file until a : is encountered
\ GETH  gets a hex digit from the file
\ GETHH  gets a byte value from the file as two hex digits
: GET ( - c )   HEXCHAR 1 HEXFILE HANDLE READ-PATH
   ABORT" Error reading file."
   0= ABORT" End of file encountered."  HEXCHAR C@ ;
: GET:   BEGIN GET ASCII : = UNTIL ;
: GETH ( - n )   GET DUP 3F > IF 9 + THEN 0F AND ;
: GETHH ( - n )   GETH GETH DENYBL  DUP CSUM +! ;

\ GETREC  reads an Intel hex record from the file.  The
\  record type is returned on the stack.
: GETREC ( - typ )   0 CSUM !  GET:  GETHH ( length )
   GETHH >< GETHH + ( adrs )  DUP XADR  0D EMIT 4 U.R
   GETHH ( type )  SWAP
   ?DUP IF 0 DO  GETHH X!+  LOOP THEN   ( data bytes )
   CSUM @ GETHH + 0FF AND ABORT" Checksum error in hex file." ;

\ LOAD  loads a hex file into the target's memory.
: LOAD   CMEM  HEXFILE PATHNAME "" .HEX" HEXFILE SET-EXT
   HEXFILE OPEN-PATH-PCB
   ABORT" Can't open file."   CR
   BEGIN GETREC UNTIL  ( continue until record type <> 0 )
   HEXFILE HANDLE CLOSE-PATH ABORT" Can't close file." ;

ONLY FORTH ALSO TALKER
COM1:  S-INIT  FAST
.( Talker is ready.) CR
```


FIGURE 3. TALKER HOST PROGRAM COMMANDS

COMMAND	FUNCTION
CMEM	selects Z8 "C" (code) memory for subsequent DUMPs and SETs.
EMEM	selects Z8 "E" (external) memory for subsequent DUMPs and SETs.
REGS	selects Z8 registers for subsequent DUMPs and SETs.

Note: many commands leave CMEM selected. When in doubt, always explicitly specify the desired memory space.

DUMP "address length DUMP" will dump a range of memory or registers. "address" and "length" are both hex values; note that they PRECEDE the DUMP command word. BOTH MUST BE GIVEN. DUMP will use the latest CMEM, EMEM, or REGS selection.

Example: EMEM 8000 50 DUMP

>>> LIMITATIONS <<<

The current version of DUMP will always dump a multiple of 16 bytes.

There is, at present, no way to abort a long DUMP.

SET "address SET" will examine memory or registers one byte at a time, and allow optional modification. Starting at "address", the program displays the address, its contents, and then awaits input. You may either enter

<CR>	to leave unchanged & advance to next address
number <CR>	to change the contents to "number" and advance
Q <CR>	to end the SET command

This is intended to resemble the Zilog "D address" command. Note that the address must PRECEDE the SET command word. SET will use the latest CMEM, EMEM, or REGS selection. All values are given in hex.

Example: REGS 0 SET

>>> LIMITATIONS <<<

An invalid "number" will abort the SET command.

In a future incarnation we will allow decimal numbers or Forth arithmetic expressions to be used for the SET values.

FILL "address length value FILL" fills a range of memory or registers with the a byte value. "address", "length", and "value" are all hex values; note that they PRECEDE the FILL command word. ALL MUST BE GIVEN. FILL will use the latest CMEM, EMEM, or REGS selection.

Example: CMEM FF00 100 55 FILL

HEX FILE SAVE AND LOAD FUNCTIONS

LOAD "LOAD filename" loads the Intel format hex file into the Z8's "C" (code) memory. The address and length are obtained from the file. Note that the filename FOLLOWS the LOAD command word.

Example: LOAD P4TEST.HEX

NOTE that LOAD will leave CMEM selected when done.

```
SAVE  "address length SAVE filename"  saves a range of "C"
      (code) memory as an Intel format hex file. "address"
      and "length" are both hex values, and must PRECEDE
      the SAVE command; the filename must FOLLOW the SAVE
      command.  All of these must be given.

      Example:        8000 300 SAVE TEST.HEX

      NOTE that SAVE will leave CMEM selected when done.

      "length" must not exceed 7FFF hex (32K bytes minus
1).

                    BREAKPOINT FUNCTIONS

The talker host program allows ten independent breakpoints
to be set in a target program. Note that all breakpoint
information is kept in the host PC.

BPn   "address BPn"  (where n is 0 to 9) sets breakpoint
      'n' at the given Z8 hex address. If breakpoint 'n'
      was already set somewhere else, the previous
      breakpoint is cleared.

      Example:        807F BP2        (sets breakpoint #2)

      "0 BPn"  (where n is 0 to 9) CLEARS breakpoint 'n'.
      If breakpoint 'n' was not set, this performs no function.

      Example:        0 BP0           (clears breakpoint #0)

      Implementation notes (Z8): a breakpoint is set by
      patching a 3-byte CALL instruction at the given
      address. The previous contents of those three bytes
      are saved inside the host PC. When the breakpoint is
      re-set to a new address, or cleared (set to address
      0), the original three bytes are put back into the
      Z8 memory.

      >>> LIMITATIONS <<<

      The breakpoint address MUST be the address of an
      instruction in the application program.

      Each breakpoint uses 3 bytes in the application
      program. So, a breakpoint should not be set (for
      example) on a RET instruction if the following two
      bytes are the beginning of another subroutine. Also,
      two independent breakpoints cannot be set less than
      3 bytes apart.

      If you reload a new version program while breakpoints
      are set, the stored 3-byte code fragments may no
      longer be valid, and clearing a breakpoint may patch
      these old fragments into the new program.  To avoid
      this, use INITBP.


INITBP clears ALL breakpoints, WITHOUT patching any code
      fragments back into the Z8's memory.

      NOTE that if you do this when breakpoints are set,
      you will leave some breakpoint CALLs patched into
      the Z8 program, with no way to find them and no
      record of their previous contents. In general, use
      INITBP only when you are reloading the application
      program.


.BP   prints a list of all the currently set breakpoints,
      and their addresses.

GO    "address GO" starts a Z8 program at the given hex
      address. The host PC will then wait for one of three events:

      1. A breakpoint is encountered.  This causes the
         message "BREAKPOINT n at: aaaa  flags=dd"  to
         appear on the screen.  "aaaa" is the address of
         breakpoint "n", and "dd" is the contents of the
         flags register when the breakpoint occurred.

         If a breakpoint CALL is encountered that does
         not correspond to a currently set breakpoint (such
         as a breakpoint left in memory by an ill-timed
         INITBP command), the message is "BREAKPOINT unknown
         at: aaaa  flags=dd".

      2. A reset (or a JP 0Ch) occurs, restarting the Z8
         talker program.  This simply causes the "Ok"
         message to appear.

         Actually, ANY character transmitted over the Z8
         UART, except an asterisk (*), will cause this to
         happen.  (The '*' character is the signal of a
         breakpoint.)

      3. ESC is typed at the PC keyboard.  This aborts the
         wait loop, but leaves the application program
         running in the Z8.

RESUME continues program execution from the location of the
       last encountered breakpoint. The Z8 flags are
       restored to the values they had when the breakpoint
       was encountered.

      Note: you must clear the breakpoint before you can
      RESUME at that address.  If you try to RESUME without
      clearing the breakpoint, you will simply trip the
      breakpoint again, instantly!


                    SPECIAL FUNCTIONS

TERM  activates a "dumb terminal" program over the COM1
      serial link. The serial port is set at 4800 baud, 8
      bits, 1 stop, no parity. Characters typed at the PC
      keyboard are sent over the serial port; characters
      received from the serial port are displayed on the
      PC screen.  Note that this is a "full duplex"
      terminal: characters typed on the keyboard are NOT
      automatically echoed to the screen.

      The ESC key ends the terminal program and returns to
      the talker host program.
```

YASBEC, from page 4

with interest to see how that works out.

The system software disk contains a full range of YASBEC system utilities and generic ZCPR3 utilities. A special copy of ZMP, the Zmodem communications program, is included. Why special? Well, it doesn't need to be installed for your terminal; everything it needs to know comes out of the ZCPR termcap. A small touch, but very handy. Several of the people who received the first round of boards were neophyte CP/Mers, and we wanted to make things as easy on them as possible.

The Documentation

The YASBEC documentation consists of a 136 page software manual, and a complete set of schematics. A hardware manual is in the works, but there's that problem with not enough hours in the day again.

And They All Lived Happily Ever After

The two original YASBEC production boards have been in constant use since the start of the year. Both prototype boards have, after a long and error-free service, been honourably retired. And stripped for their parts.

Creating the YASBEC computer has been an interesting experience. We've learned more than anyone would ever possibly want to know about the idiosyncrasies of the Z180 processor, and that's a lot. We've also learned about the dan-

See YASBEC, page 56

Ampro, from page 14

pable of taking tools to their boards, but I'll avoid the temptation to digress into a tutorial on the subject and offer this advice instead: ask a friend to perform the surgery. It should require 5 minutes' time (once the board is on the bench) and cost in the neighborhood of $5 for parts.

To provide the required inversion I elected to use an 74LS00 quad NAND gate. NANDs or NORs are more useful than inverters as spare devices, hence my personal preference (old habits die hard). I've drawn the changes using the 'LS00, but the choice of device is yours.

Figure 1 shows the a portion of the Ampro CPU 1B schematic in the area to be changed with the modification completed. The reference designators differ between the 1A and 1B CPUs. CPU 1A reference designators for the same devices are in brackets.

There are four operations to be performed:

1) Add the new 'LS00 at U1 [none].

2) Cut the trace between pin 17 of the DART (U15 [U7]) and pins 4 and 5 of U9 [U8], a 75188 (or 1488) RS-232 driver.

3) Jumper pin 30 of the DART (W/RDYB*) to pins 1 and 2 of the 'LS00.

4) Jumper pin 3 of the 'LS00 to pins 4 and 5 of U9 [U8].

If you have the Ampro CPU 1B, then all you must do is use the already provided IC location (designated U1) for the new 'LS00. It's already got power and ground routed to it, so all you need to do is attach jumpers. Use of a socket rather than installing the new chip directly is strongly suggested.

If you're modifying the Ampro CPU 1A, you'll have to attach the new chip some other way. I prefer to piggy-back on an existing chip, soldering pins 7 and 14 (GND and Vcc) to the those of the 'host' chip, bending the piggy-backer's other pins out horizontal to the board, and soldering to those 'flying' pins. If you try this, make sure that as much air space as possible is left between the chips to allow for cooling.

No matter what device you use to achieve the required inversion, remember that unused inputs of the device should not be left floating. Floating input pins can cause a variety of problems on a board, noise (which can cause spurious incorrect operation of some devices) being the most insidious. The preferred way of 'tying off' unused inputs is to attach them to Vcc via a current limiting resistor (4.7K is typical for LSTTL). According to the manufacturers' documentation, LSTTL inputs can be tied directly to Vcc without the current limiter, but use of the resistor is still typical practice. I prefer to use the current limiter, but did not want to add any more parts to the board, so I chose, instead, to tie all of the unused input pins on the 'LS00 (pins 5,6,9,10,12 and 13) to GND. This causes more power to be consumed (that's why pulling up to Vcc is preferred), but is acceptable on a small scale. If you choose to use the pull-up method, individual resistors aren't required if the size of the pull-up is adjusted. A single resistor in the 600-700 Ohm range can handle all of the unused inputs on the device you use.

Modifying the Cable

For this portion of the changes you're on your own! Ampro did at one time (and may still) sell cabinets and cables, but many Ampro systems (like mine) have been assembled by their owners from the CPU board and whatever was available or handy. Indeed, the cabling arrangements on my two Ampros differ! The best I can do is state what changes need to be made, suggest where to make them, and state what the final configuration needs to be. You'll have to decide how and where to make the changes.

The cabling needs to be modified to deliver serial port A's handshaking lines (HSIA and HIOA) to the modem. It's probably easiest to make the changes close to the Ampro before any cabling connecting the modem to the board. Assuming that you're using the system with a communication overlay that uses serial port B's handshaking lines for the modem's DCD and DTR lines, the final cabling arrangement is as follows:

AMPRO CONNECTOR/PIN	AMPRO SIGNAL NAME	MODEM PIN (DB25)	MODEM SIGNAL NAME
J4 / 1B	PRT	1	Chassis Ground
J4 / 2B	GND	7	Signal Ground
J4 / 3B	TXDB	2	Transmitted Data
J4 / 4B	HSOB	20	Data Terminal Ready
J4 / 5B	RXDB	3	Received Data
J4 / 6B	HSIB	8	Carrier Detect
J3 / 4A	HSOA	4	Request to Send
J3 / 6A	HSIA	5	Clear to Send

The column labeled Ampro CONNECTOR/PIN refers to the connector and pin on the Ampro board, itself (shown in Figure 1), and does not refer to any cable connector.

Modifying the Communications Software

Your communication software must now be modified to handle the new controls. Actually, the changes are not major. Users of communication programs like MEX or IMP need to make a few additions to their overlays. Sysops (BYE users) have got a few extra changes to make. I've opted to present the BYE changes. MEX and IMP users should be able to find the corresponding points to modify in their overlays.

In the supplied code listings, additions and changes are in lower case. Original code is in upper case. I've left enough lead-in text and/or labels to allow the areas to be modified in the Ampro insert or BYE program to be located.

First, the DART must be programmed to use the W/RDYB* line as a DMA request signal for the receiver. This should be incorporated into the initialization of the serial port. Four instructions need to be added to the initialization code. Listing 1 shows the changes from my BYE insert for the Ampro.

Next, the transmitter status routine must be modified to make sure that the modem is capable of accepting a character when the DART is able to. This requires the sampling of the modem's CTS signal. The changes to the BYE insert are in Listing 2. MEX and IMP users must make slightly more involved changes (the spaces for the status routines are fixed in size and A-register significance on return to the calling routine may differ).

There is no corresponding change to make to the receiver routine. The hardware changes handle RTS to the modem without further software intervention.

Next, the Ampro insert must be expanded from its original maximum baud rate of 2400 to 9600. This involves adding the labels SET4800 and SET9600 (and the code to program the hardware for those baud rates). Listing 3 shows my new baud rate selection routines. Note that there's a section of constants included in Listing 3 which must be placed in a different section of the BYE program. They're presented with the code in the same listing for clarity's sake. These constants replace the equates BD300, BD1200, and BD2400 in the BYE insert. IMP and MEX overlays for the Ampro I've seen al-

```
Listing 4:
BYE modifications for USR result codes

;
; Get next character if first was a '1'
;
      CALL   CHECK1      ;Is it a 1, 10 or 11?
       ENDIF             ;B5IM
;
       IF    B5IM AND PRGRSS
      CALL   RCDISP      ;Show RC to local terminal
       ENDIF             ;B5IM AND PRGRSS
;
       IF    B5IM
      cpi    0ffh      ;Error?                    <-INSERT
      jz     set3      ;If so, must have been a '1'<-INSERT

      push   psw
      call   eatall
      pop    psw

      CPI    '0'
      JZ     SET24       ;For Vadic and Hayes, 10=2400 bps
      cpi    '3'         ;For 9600 non-ARQ <-INSERT
      jz     set96       ;Go set baud rate <-INSERT
      cpi    '5'         ;For 1200 ARQ     <-INSERT
      jz     set12       ;                 <-INSERT
      cpi    '6'         ;For 2400 ARQ     <-INSERT
      jz     set24       ;                 <-INSERT
      cpi    '7'         ;For 9600 ARQ     <-INSERT
      jz     set96       ;                 <-INSERT
      cpi    '9'         ;For 4800 ARQ     <-INSERT
      jz     set48       ;                 <-INSERT
      cpi    '8'         ;For 4800 non-ARQ <-INSERT
      jz     set48       ;                 <-INSERT
;
      JMP    SET3        ;Was 1 (300 baud)
;
MDR2: push   psw
      call   eatall
      pop    psw
      CPI    '5'         ;1200 bps?
      JZ     SET12       ;Yes
      CPI    '6'         ;Some modems use 6
      JZ     SET24
      CPI    '9'         ;Or 9
      JZ     SET24       ;For Connect-2400
       ENDIF             ;B5IM
```

ready support these rates, so you'll probably not need to change yours.

The last required changes are to the BYE program itself (as opposed to the Ampro insert). These changes are required to allow BYE to interpret the USR's result codes so that the proper baud rate can be selected. These changes have been circulated around the BBS community in a file named BYE9600.FIX. I cannot credit the originator of the file—there was no authorship claim in the file. I've added a few additional delays in my implementation of the changes, which appears in Listing 4.

There are a few additional changes to BYE that don't quite fall into the 'required' category—but which I've found helpful and/or necessary for the configuration I use:

1) When using the DOATZ control, I've found that the modem takes much longer than the 1 second processing time allowed in the program to respond to the ATZ command. Through experimentation, I've found that an additional 2 second delay must be installed after the ATZ string is sent. Without the delay, commands sent after the ATZ may be ignored.

2) The USR will ignore commands while its DTR input is low. If you use the OFFHK control in BYE, you'll have to assure that the DTR control is only used to hang up the phone, and that it is not removed for extended periods of time. Specifically, it must be *on* when command strings are being sent to the modem. I had to modify my Ampro insert slightly for this. Changes can be found in Listing 5.

Setting Up the Modem

To complete the job, a few commands need to be sent to the USR to cause it to operate with hardware handshaking and to avoid baud rates in excess of 9600. Only a few changes to the factory defaults must be made. No changes to the factory default DIP switch setting are required.

With your modem program in terminal mode, enter the following string:

```
AT&FS27=128S34=3&H1&R2&W
```

The elements of this command string are:

```
&F       - Start with the factory defaults
S27=128  - Disable 7200 (and 12000, 14000) result codes
S34=3    - Disable rates in excess of 9600
&H1      - Enable hardware transmit flow control
&R2      - Enable hardware receive flow control
&W       - Write settings to Non-Volatile RAM
```

The two register commands bear some discussion due to their ambiguous and/or obtuse descriptions in the modem documentation:

When the most significant bit of register 27 is set (S27=128), the result codes for connect at 7200 (and 12000 and 14000) are not sent by the modem. These result codes are two digit codes beginning with '2', which will not be processed by BYE or most CP/M modem programs—they all expect 'connect' result codes to begin with a '1'. A baud rate of 7200 is not supported as a computer-to-modem rate either by the USR or by the Ampro (only as modem-to-modem rate), so it's use isn't necessary in the first place. Regardless of the state of this control, when a 7200 baud connection is established, a computer-to-modem baud rate of 9600 would have to be used. With this control in force, a 'CONNECT 9600' result code will be sent if a modem-to-modem rate of 7200 is initially established.

Setting the least significant 2 bits of register 34 (S34=3) disables both V.32*bis* and USR's proprietary version of V.32*bis*. This setting disables high speed modulation and assures that the modem will not attempt to establish a connection above 9600 baud. The documentation is somewhat murky when describing these controls, but a bit of research yields the information that this setting does not disable v.32*bis*—it merely keeps the modem from trying to go above 9600 baud when establishing a connection.

```
Listing 5:
Modification to AMPRO BYE insert for use of
DTR and OFFHK control

MDQUIT:  IF    IMODEM       ;If using a smartmodem
        CALL   IMQUIT       ;Tell it to shut down
         ENDIF              ;IMODEM
         if    imodem and (nodtr or offhk)
        ret
         endif              ;imodem and (nodtr or offhk)
```

Other Concerns, Problems and Tricks

Once full hardware handshaking is implemented, one is free to use a rather interesting feature of the USR: fixed com-

puter-to-modem rates. This feature is accessed by the &Bn commands. By using the &B1 command, for example, the computer-to-modem baud rate will not change regardless of the rate the connection is established at. I have experimented with this, hoping to simplify my BBS operation by always using 9600 baud between the Ampro and the USR. The feature works well (and, coincidentally, provides a fine demonstration of transmitter flow control in operation). However, I advise against it's use on a BBS. Since the modem buffers a good deal of data, a person calling into a system using this feature finds it virtually impossible to use XON-XOFF flow control on data typed from the system to him. While the caller is receiving page 'n', the computer is sending page 'n'+1, and when he presses control-S (XOFF) to halt the steam of data, the key seems to have no effect! Those who only call out on their machines may, on the other hand, find this feature worth playing with. However, data may be buffered in the modem when this feature is used, causing file transfer problems.

Buffering of data in the modem tends to make some data transfer programs very unhappy. Consider what goes on during a transfer using MEX or IMP: When a block of data is sent (via XMODEM protocol, for example), the modem program expects an acknowledgment from the machine on the other end within a certain time. If that acknowledgment is not received in time, a time-out is declared, and the block is re-transmitted. Suppose that, because of error control re-transmissions of a previous data block or because of a difference between your computer-modem baud rate and the one being used on the phone lines, the last data block from your modem program is still sitting in the modem's buffer. From your computer's point of view, the data block was sent. The time-out timer is running, when, in fact, the message hasn't had a chance to arrive at the other modem. If your machine declares a time-out and sends the block again, the situation can rapidly deteriorate. Imagine what the receiving computer will make of the situation when it (eventually) gets the same block twice!

I've experienced similar problems in the past when using PC PURSUIT. The service's packet delays, large when system usage is high, can cause outgoing data to be buffered in the local PCP computer, making modem transfers difficult. My solution to the problem was to re-configure my IMP overlay, specifying that my system's clock was an 8 MHz rather than the 4 MHz it actually is. This had the effect of doubling all time-outs in the program, which seemed to get me past most of the problems.

For the USR, buffering will be pretty much avoided if the computer-to-modem baud rate matches the modem-to-modem rate. Buffering will still occur if the modem 'falls back' to a lower baud rate due to poor line quality or if the modem has to re-transmit blocks as part of the error correction scheme. If buffering becomes a problem, try changing the clock speed in your modem overlay. This will be preferable to disabling fall-back or turning off error control—these features help to compensate for poor line quality and save you from getting even more errors.

Some experimentation with the Xn result code option commands may be required for your modem program to work properly with the USR. Since CP/M modem programs like MEX and IMP predate the introduction today's high speed modems, they can't be expected to handle all of result codes returned by those modems. Begin experimenting with the ARQ result codes disabled (&A0).

Closing Remarks

The decisions and implementation presented above certainly do not constitute the only way to solve the technical problems of hooking the USR to an old 8-bitter. There are probably a dozen of adequate solutions for the Ampro, alone! I've tried to add sufficient general discussion so that the material will be helpful for those who may want to attempt a similar project with a different high speed modem or different host machine. Hopefully, that goal was met.●

Stepped Inference, from page 45

cycle) must be shorter than the time between interrupts. With only one activator (one inference cycle per interrupt) this is not as critical. With more than one activator (more than one inference cycle per interrupt cycle) timing becomes very important.

Application

Stepped Inference is not limited to motor control. It can be applied to anything from communication protocols to high level intelligent scheduling mechanisms. The recommended application is to connect the engine to a timer interrupt or other regular source of interruption as was done in the last article. This will make prediction of inference latency as it relates to the interrupt interval feasible.

With the stepped inference model, you may envision a system with many axes. You would assign the axis feedback monitoring responsibilities to a very quick regular interrupt, say a timer whose interval is two milliseconds. Every two milliseconds the timer interrupt fires, and all of the axes' encoders are read and updated by a small set of rules. Each encoder update assigned to the axis' MCB (motor control block). You would assign the motor movement control rules to a slower regular interrupt, say a timer whose interval is thirty two milliseconds. Every thirty two milliseconds the motor rules decide what to do to a motor based on what the MCB's state is. Finally you would assign a set of scheduling rules with their own feedback (antennae and the like for system feedback and obstacle avoidance) to a slow regular interrupt, say one quarter of a second. This set of rules would avoid collision, decide what the system of axes should do next and handle the coordination of the axes.

This is only a small possibility; many other applications abound! Stepped inference brings a lot of intelligence very close to the metal. In a forth system, and the flexibility associated with it, stepped inference becomes a very powerful tool for embedded controls.

This concludes the series on stepper motor control. Next I want to delve into the application of multiaxis systems, their coordination and the like. We're heading down the path of autonomous control. Ultimately we will develop an autonomous creature that in a primitive way can react to the world though its own learned behavior.

"It would be great if they could think on their own, dad!!!" ●

```
            if( n == 0 ) break;
        }

        return n;
}

static void check_drive_type() {
        int             n;
        union REGS      regs;
        struct SREGS    sregs;
        unsigned long   l;

        sregs.es = sregs.ds = getDS();

        regs.h.ah = 0x08;       /* get drive parameters */

        int86x( 0x13, &regs, &regs, &sregs );
                                        /* invoke BIOS */

        printf( "Number of drives is %d\n", regs.h.dl );
        printf( "Max. head-side number (?) is %d\n", regs.h.dh );
        printf( "Max. cyl./track number (?) is %d\n", regs.h.ch );
        printf( "Highest sector number (?) is %d\n", regs.h.cl );

        printf( "Type of drive %d is: ", drive_id );

        sregs.es = sregs.ds = getDS();

        regs.h.dl = drive_id;   /* drive ID */

        regs.h.ah = 0x15;       /* get drive type */

        int86x( 0x13, &regs, &regs, &sregs );
                                        /* invoke BIOS */

        n = regs.h.ah;          /* get drive type */

        switch( n ) {
        case 0:
                printf( "drive not present\n" );
                break;
        case 1:
        case 2:
                printf( "floppy drive\n" );
                break;
        case 3:
                l = ((( unsigned long ) regs.x.cx ) << 16L )
                        + ( unsigned long ) regs.x.dx;
                printf( "hard drive, %lu sectors total\n" );
                break;
        default:
                printf( "unknown type %d\n", n );
        }
}

/* Show partition table records
        0       Boot indicator 0x00 = no, 0x80 = yes
        1       Start head
        2       start sector (6 lower bits)
                hi bits of start cylinder (2 upper bits)
        3       start cylinder lo 8 bits
        4       system ID
                0x00 = none
                0x01 = PRI DOS (12)     0x04 = PRI DOS (16)
                                        0x05 = EXT DOS
                0x02 = XENIX            0x03 = XNX OLD
                0x06 = DOS BIG (?)
                0x07 = HPFS
                0x08 = AIX
                0x0A = Opus
                0x51 = Novell?          0x64 = Novell
                0x52 = CP/M?            0xDB = CP/M
                0x63 = 386/IX
                0x75 = PC/IX
                0x80 = Minix old        0x81 = Minix
                0xFF = bad blocks
        5       end cylinder
        6       end sector (6 lower bits)
                hi bits of end cylinder (2 upper bits)
        7       end cylinder lo 8 bits
```

Partition, from page 26

01 00 00 00 in this field (the data is stored least-significant byte first).

Partition size: This is another 4-byte unsigned integer giving the size of the partition in sectors.

Note that the term "sector" here is a logical sector containing of 512 bytes. In the discussion of the BPB, it was noted that the DOS operating system internally can handle smaller sizes, but not a larger size. It appears that the DOS and hard disk boot logic in PCs cannot tolerate any size other than 512. In non-PC environments, a sector size of 1024 is usually used for MFM media, including hard disks. If a sector size other than 512 is to be used on a PC, the blocking and deblocking must be done in the lower levels of the BIOS.

Typical partition table

Let's take apart a typical partition table for an AT with a 40-megabyte hard disk. The drive has 1024 cylinders and five heads. Each track has 17 512-byte sectors. The hard disk is broken into three logical drives, C:, D: and E:. As mentioned before, the first drive is by itself in a primary DOS partition, and the other two are together in an extended DOS partition. Here's a dump of the partition table in hex:

```
01BE: 80 00 02 00 01 04 50 11
      01 00 00 00 F8 5A 00 00
01CE: 00 00 41 12 05 04 D0 FF
      FA 5A 00 00 04 F9 00 00
01DE: 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00
01EE: 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 55 AA
```

In the first partition, the boot indicator is set to 80 hex. The start head is zero. The start sector is 2, and the start cylinder is zero. The partition type is 01, Primary DOS with 12-bit FAT. The end head is 4. The end sector is 16, computed as follows: 50 hex, take the 6 least-significant bits by ANDing with hex 3F, giving 10 hex or 16. The end cylinder is 273, computed as follows: Take the two most-significant bits of the previous byte, 50 hex, by ANDing with hex C0, giving 40 hex or 64; multiply by four, giving 256; then add the next byte, 11 hex or 17, giving 273. The relative sector is 1; the size in sectors is 5A F8 hex or 23,288 (11.6 megabytes).

Let's check that value. Five heads times 274 cylinders times 17 sectors per cylinder is 23,290 sectors, minus one sector for the partition table, giving 23,289. But apparently DOS has a prob-

lem with the odd number of sectors, so we round to 23,288.

On the second partition, the calculations are left as an exercise for the reader. The partition starts at head 0, sector 1, cylinder 274. It's type 05, Extended DOS with 12-bit FAT. The end head is 4. The end sector is 16. The end cylinder is 1023. The relative sector is 23,290, as we would expect; the number of sectors is 63,748 (31.8 megabytes).

The program

Provided with this article is a program, READPART.C, which will allow you to examine your own partition table. It is written for Datalight C, but should easily adapt to any common PC C. Be sure to compile it without aligning structure elements.

```
        8       relative sector (from begin of drive) (4 bytes)
        12      size in sectors (4 bytes)
*/

static void show_partition( int i_part, unsigned char *p_part_desc ) {

        printf( "Partition %d:  Boot indicator: 0x%02x  System ID: 0x%02x\n",
                i_part,
                p_part_desc[ 0 ],
                p_part_desc[ 4 ] );

        printf( "Start Head: %2d  Sector: %2d  Cylinder: %4d\n",
                p_part_desc[ 1 ],
                ( p_part_desc[ 2 ] & 0x3F ),
                p_part_desc[ 3 ] + (( p_part_desc[ 2 ] & 0xC0 ) << 2 ));

        printf( "  End Head: %2d  Sector: %2d  Cylinder: %4d\n",
                p_part_desc[ 5 ],
                ( p_part_desc[ 6 ] & 0x3F ),
                p_part_desc[ 7 ] + (( p_part_desc[ 6 ] & 0xC0 ) << 2 ));

        printf( "Relative sector: %ld  Total sectors: %ld\n",
                * ( unsigned long * ) &p_part_desc[ 8 ],
                * ( unsigned long * ) &p_part_desc[ 12 ] );
}

/* end of readpart.c */
```

Given that head start, perhaps an astute reader would like to produce an improved FDISK program. Others may take advantage of the information to develop hard disk drivers for other operating systems, such as the Z-system or a homebrew OS, which shares a hard disk with a PC. Still others may take this scheme as a basis for an improved partitioning scheme for computers of the future.

New CPU technology which is coming out today is so powerful as to be mind-boggling. With hardware available today, it is possible for a single computer to simultaneously run several CPUs and several operating systems at once. For example, a Z280 running Z-system, or a Harris RTX-2000 running Forth, could reside on a coprocessor board in a PC running DOS; the coprocessor accesses its own partition by means of a small TSR program on the PC.

To make these kinds of neat tricks happen, we've got to share our information and technology. That's why *TCJ* is here!●

Editor, from page 2

fraction of the information we present. You further understand that being exposed to this information is, in itself, a valuable lesson. How can one seek answers when one doesn't know the question? As time goes by, more and more of our articles will make sense to you. One day, you will write one, and teach the next generation. Hang in there. Ask questions. Each author gives his or her address and if not, feel free to send letters here. I will pass them on for you.

Our interest in CP/M results from a collective desire to learn rather than buy other people's knowledge. CP/M and its current day successor, Z-System, represents a full fledged operating system in 8k of code. You can get source to every part of the system and it is small enough to sink your teeth into. Hardware to run this can be bought at flea markets for under $50. If you blow something up (though I haven't heard of this happening), what have you lost? Get another used Kaypro and try again. With the risks so low and the field of knowledge so rich, you can run amok with hardware and software projects. Everything you learn will carry with you as you move to larger systems.

One reason I have always liked *TCJ* (to newcomers: I took over as editor at the beginning of this year. Rather like the fellow selling electric razors, I liked the journal so much, I bought the company!) is that we have a cross section of several disciplines here. None of us are involved in everything in *TCJ* but all of us have something to offer. I like seeing what other groups are up to. Perhaps they have something to offer me. Perhaps I should consider joining up with them. Perhaps you should, too. It is important to keep an open mind. As deadset against the appliance approach to computing that I see MS-DOS represents, I find some things done on that side of the house interesting, even exciting. The work done with PCED caused Rob Friefeld, the author of LSH, to jump back into his source and add some features. We all learn from each other.

This can all be summed up with one thought: The spirit of the individual made this industry. *TCJ* will never forget that.

Neither Rain Nor Sleet....

We had a bit of a fiasco getting the foreign copies mailed last issue. The post office told me that we had grown to the point where we needed a postal meter. In fact, they showed me their rule book. Any self-respecting post of-

---

YASBEC, from page 51

gers of designing in your RS-232 converter backwards (toasted chip, anyone?), not decoding your chips selects correctly (code that works fine in one location, but won't work if you move it twenty bytes), and a whole lot of other things that both of us would prefer to forget. But it's been fun. And there's more to come. Stay tuned!●

Reader Survey

Help us serve you better by taking a moment to fill out this survey. We will use this information in planning future issues of *The Computer Journal.* You may prefer to photocopy the survey rather than to deface your copy.

Remember that as a journal, *TCJ* takes its editorial direction from its readership. Your input is important to us.

Please mail completed questionnaires to *The Computer Journal,* PO Box 12, S. Plainfield, NJ 07080-0012

What are your interests (check all that apply):

___ Embedded Controllers
___ Robotics
___ Microprocessors (identify which)
___ Home Control
___ Interfacing
___ Communications
___ LANs and Connectivity
___ 8-Bit Operating Systems
  ___ CP/M
  ___ Turbo-DOS
  ___ Z-System
  ___ other (identify)
___ 16-bit Operating Systems
  ___ MS-DOS
  ___ Minix, Concurrent
  ___ Unix
  ___ other
___ Languages
  ___ Forth
  ___ Assembler (identify processor)
  ___ C
  ___ Pascal
  ___ Modula-2
___ Programming and Algorithms

Are you more interested in articles on:
(A) Software (B) Hardware (C) Both

If you answered "Both", what is the perfect mix you would like to see? Answer as a percentage (e.g. 40% Software, 60% Hardware):
___ Software ___ Hardware

Would you like to contribute an article? If yes, tell us briefly what the topic would be. Also, please be sure to give us your name so that we may get back to you.

_____
_____

What kind of computers do you use?

_____

Do you use computers in your employment?
___ Yes ___ No

If you answered "Yes" above, is your computer usage at work related to the topics we cover in *TCJ*?
___ Yes ___ No

Do you like having source code and schematics printed or would you rather these be available for sale on disk and leave the space free in the journal for more text?
___ Keep the listings ___ Sell by disk

Would you recommend *TCJ* to a friend or colleague?
___ Yes ___ No

If you answered "No" above, why not?

_____

If you answered "Yes" above, are you aware of the "Sponsor a Friend" policy? ___ Yes ___ No

How long have you been a reader of *TCJ* ? ___ years

TCJ has a new editor as of the first of this year. We have been working on many aspects of the journal since then. On a scale of 1 to 5 (1 being very bad and 5 being outstanding), tell us how you feel about the changes. ___

What directions would you like to see the journal go?

_____
_____
_____
_____
_____

Optional:
What is your age? ____
What is your sex? ___ Male / ___ Female
What is your occupation? _____
What is your annual income?
___ Less than $20,000
___ Between $20,000 and $30,000
___ Between $30,000 and $40,000
___ Between $40,000 and $50,000
___ Over $50,000

---

fice has a selection of rule books. This one said they didn't need to handle more than six transactions at the counter for any single customer. I said I have been happily married for years, but this didn't phase them.

It was time to get a meter.

Now, I don't mean to tell stories on any particular company. We can all have our problems. If I tell you that the postage meter company promised the unit in three to four weeks and took nearly two months, and since there is only one postage meter company in the United States, I would be telling you which company screwed up. So, I won't. What I will say is that the foreign issues were mailed the morning after the meter arrived. Shipments of back issues and other mail was similarly delayed. My apologies. Meanwhile, I reserve my own right to mess up.

A new wrinkle seems to be appearing as regards to Canadian subscriptions. Evidently, it has been taking about a month and a

See Editor, page 60

Z-Node List #63

Last Update June 27, 1991 by Ian Cottrell

This list includes information about accessing the Z-Nodes using the two low-cost data services, PC-Pursuit (PCP) and StarLink (SL). For nodes accessible by PCP, the city code and maximum data rate are given. Where known, the StarLink code is given (all Starlink nodes support 2400 bps). An asterisk with the code indicates that the call to the Z-Node may incur local toll charges. If you know the SL code for any nodes that do not have a code listed, I would appreciate it if you would send me that information.

Well, you win some and lose some. Node 13 in Fergus, ON is temporarily down while Larry Moore relocates and gets things re-established. Also down this time are node 66 (Dave Van Horn), node 21 (Dick Roberts) and node 4 (Ken Jones). But on the plus side, Ben Grey reports node 8 is up and running again in Portland and Ian Cottrell joins the list as node 5 in Ottawa, ON, Canada.

Report any changes or corrections in a message to Jay Sage on Z-Node Central (#2) or Z-Node #3 in Boston (or by mail to 1435 Centre St., Newton Centre, MA 02159-2469).

A '' in the first column indicates that the node uses a 9600 bps modem.*

	NODE	SYSOP	CITY	STATE	ZIP	RAS Phone	PCP	SL	Verified
	Z-Node Central								
	2	Al Hawley	Los Angeles	CA	90056	213-670-9465	CALAN/24	3173*	04/20/91
	Satellite Z-Nodes:								
	UNITED STATES								
	3	Jay Sage	Newton Centre	MA	02159	617-965-7259	MABOS/24	8796	06/22/91
	4	Ken Jones	Salem	OR	97305	503-370-7655		3174	(down)
	6	Robert Dean	Drexel Hill	PA	19026	215-623-4040	PAPHI/24	9581	06/22/91
	7	Dave Trainor	Cincinnati	OH	45236	513-791-0401		1785*	06/22/91
	8	Ben Grey	Portland	OR	97229	503-297-0741	ORPOR/24	9164	07/05/91
*	9	Roger Warren	San Diego	CA	92109	619-270-3148	CASDI/24	9183*	06/22/91
	10	Ludo VanHemelryck	Mill Creek	WA	98012	206-481-1371	WASEA/24	9170*	06/22/91
	11	Carson Wilson	Chicago	IL	60626	312-764-5162	ILCHI/24	8257*	06/22/91
	12	Lee Bradley	Newington	CT	06111	203-665-1100	CTHAR/24	9128	(down)
	15	Liv Hinckley	Manhattan	NY	10129	212-489-7370	NYNYO/24	1059*	10/25/89
	16	John Anderson	Colonie	NY	12205	518-489-1307		9192	10/29/90
	17	Bill Biersdorf	Tampa	FL	33618	813-961-5747	FLTAM/24	5518	(down)
	21	Dick Roberts	S. Plainfield	NJ	07080	908-757-1491	NJNBR/24	3319	(down)
	32	Chris McEwen	S. Plainfield	NJ	07080	908-754-9067	NJNBR/24	3319	06/22/91
	33	Jim Sands	Enid	OK	73703	405-237-9282		10816	06/22/91
	36	Richard Mead	Pasadena	CA	91105	818-799-1632		2940*	06/22/91
	45	Robert K. Reid	Houston	TX	77088	713-937-8886	TXHOU/24	4562*	06/22/91
	58	Kent R. Mason	Oklahoma City	OK	73107	405-943-8638		9165*	(down)
	65	Barron McIntire	Cheyenne	WY	82007	307-638-1917		4213	06/22/91
	66	Dave Vanhorn	Costa Mesa	CA	92696	714-546-5407	CASAN/24	9184*	(down)
	77	Lindsay Haisley	Austin	TX	78745	512-259-1261		1306	06/22/91
	78	Gar K. Nelson	Olympia	WA	98502	206-943-4842			06/22/91
	81	Robert Cooper	Lancaster	CA	93535	805/949-6404		5991*	11/06/89
	CANADA								
	5	Ian Cottrell	Ottawa, ON		K2G 0T7	613-952-2289			06/27/91
*	13	Larry Moore	Fergus, ON		N1M 3H7	519-843-7314			(down)
	20	Brian Grover	Burnaby, BC		V5C 2Y3	604-299-0935			03/11/90
	40	Greg Kopp	Winnipeg, MB		R2C 1J4	204-224-1282			01/01/91
	AUSTRALIA								
	50	Mark Little	Alice Springs	NT	5750	61-089-528-852			(????)
	62	Lindsay Allen	Perth	WA	6153	61-9-450-0200			07/01/89
	GERMANY								
	51	Helmut Jungkunz	Munich, GERMANY			NBBS 08165/60041			03/30/91

TCJ The Computer Journal Market Place

Editor, from page 57

half for the copies to arrive. I am not sure what is going on here. Until we get to the bottom of it, I would appreciate any feedback from you kind folks north of the border. One person reported that it took three months for his copy of issue 49 to appear! And I thought the US Postal Service held title to the term "Snail Mail!" Maybe the two groups are working together on this, and what we have is the postal equivalent of Critical Mass.

Good Folks To Know

The Rochester Conference on Automated Controls is being held as I write this. I've been in contact with Larry Forsley of the Forth Institute, which has been putting this conference on over the years. He sent a copy of last year's proceedings. This conference is not for the casual observer; the list of speakers looks like a Who's Who in the Forth world. I would like to make next year's conference, just to rub shoulders. For this year, I had to satisfy myself with providing a copy of our last issue for each participant.

I was on the telephone with Randy Dumsey at New Micros the other day. Word has it that NMI is hosting three gentlemen from the Soviet Union for the next few months. I had been mulling over an idea for an article: could they tell us how the working environment differs for programmers in the two countries? We have all been hearing comparisons of "computing" between the two nations. But "computing" implies just machinery. What about the people? Randy had the three in the room with him and turned the speaker phone on. Unfortunately, the connection was not clear enough for me to jot their names down, but they all liked the idea.

There are two schools of thought towards adding capabilities. You can keep stacking more chips, boards and peripherals on a system until you blow the power supply and your budget, or you can take a good strong look at the system you have and get the last ounce of efficiency from it. The latest, greatest achievements in American computing follow the first path; Windows needs two megabytes of RAM and Unix wants four. It is a rare programmer that worries about the size of code he writes. But what if your project requires a given level of performance and buying the hardware makes your product unprofitable? Moving from 8-bits to 16 can quadruple the price of your components. The extra cost of efficient coding is spent once but the cost of additional hardware is repeated with each unit.

Since programmers in the Soviet Union have limited hardware resources, they have to get more from less. This forces programming discipline. As I see it, their shortage of hardware is putting them in a position to whip our pants when they do get access to state-of-the-art machinery. When they combine their efficient code with our efficient equipment, we will be left standing in the dust.

We stand to learn from these fellows.

In his column, Jay mentions the *Z-Letter*. I made a passing comment on this a few months ago. At that time, I had never seen a copy. I have now, and I must say that I was impressed. David McGlone does a fine job with it.

A group that Jay didn't mention, because he didn't know of them, is the *Morrow-Atlanta Users Group*. They produce what may be the best user group newsletter I have seen. In fact, I was so impressed that I am sending a sample of this issue to each member. Ask for information by writing to Mr. Harold Arnovitz, Editor, *Mor-Atlanta News*, 1259 Kittredge Ct NE, Atlanta GA 30329.

Late breaking news: Lee Bradley just announced that he is ceasing publication of Eight Bits & Change. We have mentioned EB&C on several occasions. Jay lists it at the end of his column in this issue. Please make a note that EB&C moves into history with the upcoming issue. We'll miss your work, Lee.

Lee will be taking his Z-Node down at the same time. But for those who step back, others step forward. Ian Cottrell has joined the ranks with Z-Node 5. And Ben Grey, a Z-Node sysop for years, has gone back on-line.

Mountain View Press

Glen Haydon has taken over at Mountain View Press and will be running it as a division of Epsilon Lyra, along with WISC Technologies. Glen is working on synthesis tools for process control and is looking at Mitch Bradley's approach to boot ROMs that will automatically configure systems.

Mountain View Press publishes books on Forth, and was chosen as a center for Forth educational materials at the recent Rochester Forth Conference.

You can contact Glen at (415) 747-0760 or write to Mountain View Press, Box 429, Route 2, La Honda CA 94020.

What We Have and What We Don't

This issue introduces four new authors. Roger Warren tells us how to modify an Ampro Little Board to handle hardware flow control. He found this necessary when he installed a 9600 bps modem. The solution he found can be applied to any machine, so if you have another brand, don't despair.

Paul Chidley and Wayne Hortensius introduce their new computer, the YASBEC. I mentioned this in passing last issue. What started as a home-brew project is quickly developing into a cult. As a result, Paul and Wayne find themselves being pushed into production. The possibilities here are exciting and we will report what develops. I hear that Hal Bower is involved in putting his banked system on the YASBEC. Perhaps we can convince Hal to tell us more on this in a later issue.

Paul tells me that they have ordered a boatload of boards. This is good news to those who have been waving checkbooks and shouting "When? Where? How much?" Be advised the line starts from the left, first come, first served.

More good news: the Calgary Crew has a color graphic video board up and running. They've been hitting each other over the head with problems relating to RFI until they discovered the flux on one type of solder was slightly conductive. So, bye-bye wavy lines, hello stable picture and new board. This story gets more and more interesting. Think they'll come up with a multi-I/O board?

Brad Rodriguez joins us with an article on the Z8. He has

TCJ On-Line

Readers and authors are invited to join in discussions with their peers in any of three on-line forums.

- GEnie Forth Interest Group (page 710)
- GEnie CP/M Interest Group (page 685)
- Socrates Z-Node 32

For access to GEnie, set your modem to half duplex, and call 1-800-638-8369. Upon connection, enter HHH. At the U#= prompt, enter XTX99486,GENIE and press RETURN. Have a credit card or your checking account number handy.

Or call Socrates Z-Node, at (908) 754-9067. PC Pursuit users, use the NJNBR outdial. StarLink users, use the 3319 outdial.

a "Talker" that takes under 300 bytes on the target system and uses a host computer for the majority of the logic. This results in having a fully-capable monitor available for an untested system in development. Have you ever wondered how you bring up a system that has never been booted before? Is this the computer equivalent to the chicken-and-egg question? Brad works us through this.

Frank Sergeant finishes up his disk aligner project in this issue and gives us an introductory look at Forth. A fair share of our readers have been asking where to start. Frank has the answers. Meanwhile, Matt Mercaldo finishes up his series on motor control with Stepped Inference. We welcome Matt as our newest contributing editor—his work with embedded control has earned his place on the masthead.

This is a good point to mention that our expansion to 64 pages is to provide additional coverage of Forth and embedded control without sacrificing our traditional areas of coverage. You are already seeing some of the new material. I am still prowling for others. Our liberal payment policy applies. (*"Payment? This is User-Supported Publishing!"*) I'd like a thrust towards practical applications. Let's see what you've been cooking up in that workshop of yours.

Rick Rodman reveals the secrets of the MS-DOS hard disk partition table. You would think that ten years after the fact, we would have ready access to this information. In fact, it takes real digging to find it and Rick comes through for us once again.

Interesting item: the BPB contains a byte to tell the machine what operating system you are booting. I hadn't known that. Adding to this, Brian Moore reports that he has successfully ported Z-System to his '386 box, running ZCPR 3.3, and Z80DOS. While CP/M has been ported before, this is the first known case of installing a dynamic Z80 operating system onto an Intel platform. He promises to tell us all about it. I'd walk barefoot from New Jersey to Oregon to pick up that article!

Brian, by the way, is the author of ARK11 for CP/M and is a sysop on GEnie.

Bruce Morgen rejoins us after a brief hiatus. His topic is using CP/M and MS-DOS machines through a Unix box. This opens many possibilities: shared peripherals, file storage on a large system. And the best part is in the title, "Cheap Connectivity." All it costs is a little cabling!

That is the good news. The bad news is that we are missing two articles I had wanted to give you. Al Hawley had a bout of bad health these last couple of months and missed this issue. His series on *Assembly Language Programming for the High Level Programmer* has been exceedingly well received. We expect Al to return next issue. Meanwhile, I wish you good health, Al. Drink plenty of fluids and watch soap operas. That always gets *me* back to the office!

Jay Sage will continue with his series on Home Control next issue. His mother passed away last month. Blessings come in many ways; she was able to spend her last days with her children. I pray my passing will be as peaceful and that I may also be surrounded by love. My deepest sympathies, Jay.

Is CP/M Dead?

Jay did get his Z-System Corner column in. He takes on the old "CP/M Is Dead" syndrome. I can relate to this. What we have here is a problem of scale. Ten years ago, a total installed base of a million units was significant. Today, millions of MS-DOS machines are shipped every month. Guesses to the remaining number of CP/Mers ranges from ten thousand to well over 100,000. Whatever the real number is, it is a fraction of one week's shipment of new PC's. Does this mean CP/M is dead? By these numbers, it was never alive! I dispute this.

We face a self-fulfilling prophesy with all this talk. For example: I sent out over a thousand copies of *TCJ* to CP/M groups all over the country a few months ago. Yes, there are that many groups. Less than a dozen responded. Why? This is the sole remaining major publication for CP/M in the world. One conclusion is that CP/M is, in fact, dead. Yet in speaking to people, I hear over and over that they are still active with CP/M but don't want to invest money because "no one else is." Great! This attitude ensures the demise of support for CP/M. (Note: we have no plans to drop CP/M. My reference is to support by others).

The future of CP/M, as with any other group, requires developing new blood. This means training, helping, guiding newcomers. As long as we have something to offer, and to anyone interested in learning, really learning, about micro computers CP/M has plenty to offer, it will survive.

One hundred thousand of anything is a lot. Open your wallet and pull out 100,000 dollar bills. Shuck 100,000 clams. Copy 100,000 disks. Why must we focus on a hundred million of some other system and say our numbers are too small?

The next time you hear someone say CP/M is dead, remember the number of people represented, including yourself.

And Thanks Goes To....

I try to take a moment in each issue to thank someone who helps make *TCJ* possible. The person I will highlight this time is in a *make-or-break* position for this publication. Without her support and assistance, I would have collapsed before putting out my first issue in January. We all joke about our spouse being our "better half." Those who know both my wife and me know how serious I am when I say this. How many wives would let their husband take over one floor of the house for his projects and then fill the living room with thousands of magazines for several days every other month? Every closet in this place contains supplies, back issues, stamps, meters, lists, letters.... She keeps a semblance of order in this dervish's domain. Pardon me for just a bit, friends, while I turn to Ester and say, *"Thanks! I couldn't do it without you."*●

Success is...

"To laugh often and love much; to win the respect of intelligent persons and the affection of children; to earn the approbation of honest critics and to endure the betrayal of false friends; to appreciate beauty; to find the best in others; to give of one's self; to leave the world a bit better, whether by a healthy child, a garden patch, or a redeemed social condition; to have played and laughed with enthusiasm and sung with exultation; to know that even one live has breathed easier because you have lived—this is to have succeeded."—*Attributed to Ralph Waldo Emerson (1803-1882)*

The Computer Journal

Back Issues

Sales limited to supplies in stock.

Special Close Out Sale on these back issues only.

3 or more, $1.50 each postpaid in the US or $3.00 postpaid airmail outside US.

Issue Number 1:

- RS-232 Interface Part 1
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an 'Epram'

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part 2
- Build Hardware Print Spooler Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 5:

- Build VIC-20 EPROM Programmer
- Multi-User: CP/Net
- Build High Resolution S-100 Graphics Board: Part 3
- System Integration, Part 3: CP/M 3.0
- Linear Optimization with Micros

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using the Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part 7
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking & Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering & Other Strange Tales
- Build an S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures & Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition & Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The Ampro Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing the CP/M Operating System
- INDEXER: Turbo Pascal Program to Create an Index
- The Ampro Little Board Column

Issue Number 24:

- Selecting & Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assemble Code for CP/M
- The C Column: Software Text Filters
- Ampro 186 Column: Installing MS-DOS Software
- The Z-Column
- NEW-DOS: The CCP Internal Commands
- ZTime-1: A Real Time Clock for the Ampro Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs. Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro LB
- Building a SCSI Adapter
- NEW-DOS: CCP Internal Commands
- Ampro 186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside Ampro Computers
- NEW-DOS: The CCP Commands (continued)
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis & Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program in Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying the CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting Your Own BBS
- Build an A/D Converter for the Ampro Little Board
- HD64180: Setting the Wait States & RAM Refresh using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM Disk to Ampro Little Board, Part 1
- Using the Hitachi hd64180: Embedded Processor Design
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro Little Board
- 3200 Hackers' Language
- MDISK: Adding a 1 Meg RAM Disk to Ampro Little Board, Part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk Parameters & their variations
- XBIOS: A Replacement BIOS for the SB180
- K-OS ONE and the SAGE: Demystifying Operating Systems
- Remote: Designing a Remote System Program
- The ZCPR3 Corner: ARUNZ Documentation

Issue Number 32:

- Language Development: Automatic Generation of Parsers for Interactive Systems
- Designing Operating Systems: A ROM based OS for the Z81
- Advanced CP/M: Boosting Performance
- Systematic Elimination of MS-DOS Files: Part 1, Deleting Root Directories & an In-Depth Look at the FCB
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII Terminal Based Systems
- K-OS ONE and the SAGE: System Layout and Hardware Configuration
- The ZCPR3 Corner: NZCOM and ZCPR34

Issue Number 33:

- Data File Conversion: Writing a Filter to Convert Foreign File Formats
- Advanced CP/M: ZCPR3PLUS & How to Write Self Relocating Code
- DataBase: The First in a Series on Data Bases and Information Processing
- SCSI for the S-100 Bus: Another Example of SCSI's Versatility
- A Mouse on any Hardware: Implementing the Mouse on a Z80 System
- Systematic Elimination of MS-DOS Files: Part 2, Subdirectories & Extended DOS Services
- ZCPR3 Corner: ARUNZ Shells & Patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System.
- Database: A continuation of the data base primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking executive.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.
- The Computer Corner

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8088 software to produce modifiable assem. source code.
- Real Computing: The NS32032.
- S-100: EPROM Burner project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System. Part 1: Selecting your assembler, linker and debugger.
- The Computer Corner

Issue Number 36:

- Information Engineering: Introduction.
- Modula-2: A list of reference books.
- Temperature Measurement & Control: Agricultural computer application.
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE.
- Real Computing: NS32032 hardware for experimenter, CPUs in series, software options.
- SPRINT: A review.
- REL-Style Assembly Language for CP/M & ZSystems, part 2.
- Advanced CP/M: Environmental programming.
- The Computer Corner.

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers.
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER.
- Information Engineering: Basic Concepts: fields, field definition, client worksheets.
- Shells: Using ZCPR3 named shell variables to store date variables.
- Resident Programs: A detailed look at TSRs & how they can lead to chaos.
- Advanced CP/M: Raw and cooked console I/O.
- Real Computing: The NS 32000.
- ZSDOS: Anatomy of an Operating System: Part 1.
- The Computer Corner.

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS: Anatomy of an Operating System, Part 2.

The Computer Journal

Back Issues

Sales limited to supplies in stock.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:

- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0•The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

Issue Number 41:

- Forth Column: ADTs, Object Oriented Concepts.
- Improving the Ampro LB: Overcoming the 88Mb hard drive limit.
- How to add Data Structures in Forth
- Advanced CP/M: CP/M is hacker's haven, and Z-System Command Scheduler.
- The Z-System Corner: Extended Multiple Command Line, and aliases.
- Programming disk and printer functions with C.
- LINKPRL: Making RSXes easy.
- SCOPY: Copying a series of unrelated files.
- The Computer Corner.

Issue Number 42:

- Dynamic Memory Allocation: Allocating memory at runtime with examples in Forth.
- Using BYE with NZCOM.
- C and the MS-DOS Screen Character Attributes.
- Forth Column: Lists and object oriented Forth.
- The Z-System Corner: Genie, BDS Z and Z-System Fundamentals.
- 68705 Embedded Controller Application: An example of a single-chip microcontroller application.
- Advanced CP/M: PluPerfect Writer and using BDS C with REL files.
- Real Computing: The NS 32000.
- The Computer Corner

Issue Number 43:

- Standardize Your Floppy Disk Drives.
- A New History Shell for ZSystem.
- Heath's HDOS, Then and Now.
- The ZSystem Corner: Software update service, and customizing NZCOM.
- Graphics Programming With C: Graphics routines for the IBM PC, and the Turbo C graphics library.
- Lazy Evaluation: End the evaluation as soon as the result is known.
- S-100: There's still life in the old bus.
- Advanced CP/M: Passing parameters, and complex error recovery.
- Real Computing: The NS32000.
- The Computer Corner.

Issue Number 44:

- Animation with Turbo C Part 1: The Basic Tools.
- Multitasking in Forth: New Micros F68FC11 and Max Forth.
- Mysteries of PC Floppy Disks Revealed: FM, MFM, and the twisted cable.
- DosDisk: MS-DOS disk format emulator for CP/M.
- Advanced CP/M: ZMATE and using lookup and dispatch for passing parameters.
- Real Computing: The NS32000.
- Forth Column: Handling Strings.
- Z-System Corner: MEX and telecommunications.
- The Computer Corner

Issue Number 45:

- Embedded Systems for the Tenderfoot: Getting started with the 8031.
- The Z-System Corner: Using scripts with MEX.
- The Z-System and Turbo Pascal: Patching TURBO.COM to access the Z-System.
- Embedded Applications: Designing a Z80 RS-232 communications gateway, part 1.
- Advanced CP/M: String searches and tuning Jetfind.
- Animation with Turbo C: Part 2, screen interactions.
- Real Computing: The NS32000.
- The Computer Corner.

Issue Number 46:

- Build a Long Distance Printer Driver.
- Using the 8031's built-in UART for serial communications.
- Foundational Modules in Modula 2.
- The Z-System Corner: Patching The Word Plus spell checker, and the ZMATE macro text editor.
- Animation with Turbo C: Text in the graphics mode.
- Z80 Communications Gateway: Prototyping, Counter/Timers, and using the Z80 CTC.

Issue Number 47:

- Controlling Stepper Motors with the 68HC11F
- Z-System Corner: ZMATE Macro Language
- Using 8031 Interrupts
- T-1: What it is & Why You Need to Know
- ZCPR3 & Modula, Too
- Tips on Using LCDs: Interfacing to the 68HC705
- Real Computing: Debugging, NS32 Multitasking & Distributed Systems
- Long Distance Printer Driver: correction
- ROBO-SOG 90
- The Computer Corner

Issue Number 48:

- Fast Math Using Logarithms
- Forth and Forth Assembler
- Modula-2 and the TCAP
- Adding a Bernoulli Drive to a CP/M Computer (Building a SCSI Interface)
- Review of BDS "Z"
- PMATE/ZMATE Macros, Pt. 1
- Real Computing
- Z-System Corner: Patching MEX-Plus and TheWord, Using ZEX
- Z-Best Software
- The Computer Corner

Issue Number 49:

- Computer Network Power Protection
- Floppy Disk Alignment w/RTXEB, Pt. 1
- Motor Control with the F68HC11
- Controlling Home Heating & Lighting, Pt. 1
- Getting Started in Assembly Language
- LAN Basics
- PMATE/ZMATE Macros, Pt. 2
- Real Computing
- Z-System Corner
- Z-Best Software
- The Computer Corner

Issue Number 50:

- Offload a System CPU with the Z181
- Floppy Disk Alignment w/RTXEB, Pt. 2
- Motor Control with the F68HC11
- Modula-2 and the Command Line
- Controlling Home Heating & Lighting, Pt. 2
- Getting Started in Assembly Language Pt 2
- Local Area Networks
- Using the ZCPR3 IOP
- PMATE/ZMATE Macros, Pt. 3
- Z-System Corner, PCED
- Z-Best Software
- Real Computing, 32FX16, Caches
- The Computer Corner

	U.S.	Foreign (Surface)	Foreign (Airmail)	Total
Subscriptions				
1year (6 issues)	$18.00	$24.00	$38.00	_____
2 years (12 issues)	$32.00	$44.00	$72.00	_____
Back Issues				
18 thru #43	$3.50 ea.		$5.00 ea.	_____
6 or more	$3.00 ea.		$4.50 ea.	_____
#44 and up	$4.50 ea.		$6.00 ea.	_____
6 or more	$4.00 ea.		$5.50 ea.	_____

Back Issues Ordered:

_____

Subscription Total _____

Back Issues Total _____

Total Enclosed _____

Name: _____

Address: _____

_____

_____

My Interests: _____

_____

Payment is accepted by check or money order. Checks must be in US funds, drawn on a US bank. Personal checks within the US are welcome.

TCJ The Computer Journal

P.O. Box 12, S. Plainfield, NJ 07080-0012

Phone (908) 755-6186

Computer Corner

By Bill Kibler

Never a slow moment these days, which has led me to some deep thinking. I have also got some work done on porting Forth. First lets tackle those deep thoughts.

To Game or Not to Game

The company I work for just had a small layoff, well small only if you weren't one to be let go. While pondering how this might affect me, I considered what really fuels the industry. While teaching I am often asked about why high tech businesses are so volatile. Usually I say something about new products and competition and all that stuff.

The other day I started on a different concept. The computer industry is based on games, computer games that is. At first there were the 8080 and 6502 as computers for the hardware hacker. When the industry really took off, however, games are what really paid those first big profits. Ten years ago some small business computers were being sold, but it was games that fueled the market place and small computers were small business.

As an industry (the computer industry that is) things really haven't changed much over the last decade. Many of the same ideas and business tactics are still in operation. In gaming you have products with short life spans, low development cost, and hopefully no support. Now you take a large company like mine and what is management's approach to their product? They may say otherwise, but when you see where the money actually goes, you see that they are trying to sell products with a short life span, hire only cheap programmers, and put very little into support either for clients or support of the product as it matures.

Now I am not saying that only the company I work for does this. I think it is industry wide. What I think does it is funding; they learned from the game side of the industry. Your money backers call the shots in most businesses. Therefore they have a short money cycle and want fast and quick profits. Get in and win before the competition beats you to the next product.

The problem is in developing new products which have very long development cycles, and need lots of support both inside and out of the company. The hardware for these products cost plenty and needs just as much support. What happens to many companies is they blow all their funds getting things going and leave little for the long haul. I once took a course in marketing and they described a products cycle. Usually a product goes like this, slow to catch on, hot and fast once the word gets out, then drops on its face almost as fast as it started. The really good companies know how to stretch the middle part by adding more features or enhancements (otherwise known as making it appear like a new game!).

"The biggest problem with making products that can grow and be enhanced is thinking ahead."

The biggest problem with making products that can grow and be enhanced is thinking ahead. You can't design a product one weekend and expect it to work well 2 years from now. It really takes skill and courage to stick to your guns and require all the items needed for a good product that can last a long time. Some of those concepts are: good modular design; good documentation; extensive preconceptualization through product development. You will also need to listen to your users and adjust to their changing needs. In short, support, more support, and plan for further support. Sounds expensive and it is.

What alternatives do you have? Hoping someone buys you out before you run out of steam? I feel that last concept is what most programmers hope for. However, quick profits and then retirement is rare in any industry. So my advice is plan, plan, and do more planning. With good planning you will find the first product falls into place and sets you up for many more options later.

Embedded Planning

When speaking of planning, I have seen a friend's Forth based embedded system. I can't say much now, as he hasn't released it officially. The main points about his project is his planning and trying to think of everything the user might want. It is one thing to just design a product and throw it out in the market and quite another thing to consider how it will be used and then provide all the possible support they might need.

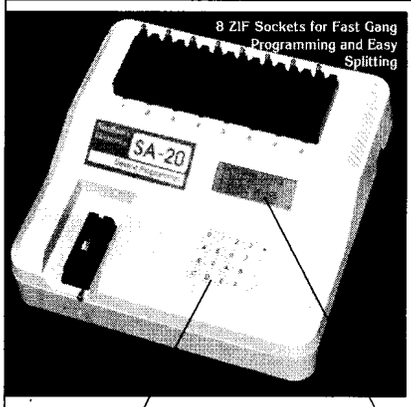
"Plan, plan, and do more planning...You will find the first product falls into place and sets you up for many more options later."

His actual boards are well designed with several options already programmed in. Their cost will make it hard not to use them. The software package will be very easy to use. It will come with several already available options, either to use or experiment with. If I had had this product when we were designing a controller for a coating process, I would have used it and saved plenty of dollars in time

See Corner, page 40

EPROM PROGRAMMERS

Stand-Alone Gang Programmer $750.00

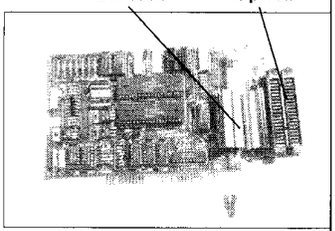


- Completely stand-alone or PC driven
- Programs E(E)PROMs
- ***1 Megabit of DRAM***
- ***User upgradable to 32 Megabit***
- ***.3/.6" ZIF socket, RS-232, Parallel In and Out***
- 32K internal Flash EEPROM for easy firmware upgrades
- ***Quick Pulse Algorithm (27256 in 5 sec, 1 Megabit in 17 sec.)***
- 2 year warranty
- Made in U.S.A.
- Technical support by phone
- Complete manual and schematic
- ***Single Socket Programmer also available. $550.00***
- Split and Shuffle 16 & 32 bit
- 100 User Definable Macros, 10 User Definable Configurations
- Intelligent Identifier
- Binary, Intel Hex, and Motorola S

Internal Programmer for PC $139.95

New Intelligent Averaging Algorithm. Programs 64A in 10 sec., 256 in 1 min., 1 Meg (27010, 011) in 2 min. 45 sec., 2 Meg (27C2001) in 5 min. Internal card with external 40 pin ZIF.

- Reads, verifies, and programs 2716, 32, 32A, 64, 64A, 128, 128A, 256, 512, 513, 010, 011, 301, 27C2001, MCM 68764, 2532
- ***Automatically sets programming voltage***
- Load and save buffer to disk
- Binary, Intel Hex, and Motorola S formats
- ***Upgradable to 32 Meg EPROMs***
- ***No personality modules required***
- 1 year warranty • 10 day money back guarantee
- Adapters available for 8748, 49, 51, 751, 52, 55, TMS 7742, 27210, 57C1024, and memory cards
- Made in U.S.A.



NEEDHAM'S ELECTRONICS
4539 Orange Grove Ave. • Sacramento, CA 95841
Mon. - Fri. 8am - 5pm PST
C.O.D.

Call for more information
(916) 924-8037
FAX (916) 972-9960

Cross-Assemblers as low as $50.00
Simulators as low as $100.00
Cross-Disassemblers as low as $100.00
Developer Packages
as low as $200.00(a $50.00 Savings)

A New Project
Our line of macro Cross-assemblers are easy to use and full featured, including conditional assembly and unlimited include files.

Get It To Market--FAST
Don't wait until the hardware is finished to debug your software. Our Simulators can test your program logic before the hardware is built.

No Source!
A minor glitch has shown up in the firmware, and you can't find the original source program. Our line of disassemblers can help you re-create the original assembly language source.

Set To Go
Buy our developer package and the next time your boss says "Get to work.", you'll be ready for anything.

Quality Solutions
PseudoCorp has been providing quality solutions for microprocessor problems since 1985.

BROAD RANGE OF SUPPORT
- Currently we support the following microprocessor families (with more in development):

Intel 8048	RCA 1802,05	Intel 8051	Intel 8096
Motorola 6800	Motorola 6801	Motorola 68HC11	Motorola 6805
Hitachi 6301	Motorola 6809	MOS Tech 6502	WDC 65C02
Rockwell 65C02	Intel 8080,85	Zilog Z80	NSC 800
Hitachi HD64180	Motorola 68000,8	Motorola 68010	Intel 80C196

- All products require an IBM PC or compatible.

So What Are You Waiting For? Call us:

PseudoCorp
Professional Development Products Group
716 Thimble Shoals Blvd, Suite E
Newport News, VA 23606
(804) 873-1947 FAX: (804)873-2154

William P Woodall • Software Specialist

Custom Software Solutions for Industry:
- Industrial Controls
- Operating Systems
- Image Processing
- Hardware Interfacing
- Proprietary Languages
- Component Lists

Custom Software Solutions for Business:
- Order Entry
- Warehouse Automation
- Inventory Control
- Wide Area Networks
- Point-of-Sale
- Accounting Systems
- Local Area Networks
- Telecommunications

Publishing Services:
- Desktop Systems
- Books
- CBT
- Format Conversions
- Directories
- Interactive Video

33 North Doughty Ave, Somerville, NJ 08876 • (908) 526-5980