The COMPUTER JOURNAL

Programming - User Support Applications

Issue Number 52 September / October 1991 US$3.95

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.

TCJ The Computer Journal

Issue Number 52 September / October 1991

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

Opinions expressed in *The Computer Journal* are those of the respective authors and do not necessarily reflect those of the editorial staff or publisher.

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.

Editor's Desk

By Chris McEwen

In the last issue, I gave word on Lee Bradley's *Eight Bits & Change*. He had announced that he was ceasing publication. Whatever problems there were have been solved. I am happy to report that *EB&C* is continuing. In fact, I have received issues since then. Sorry, Lee.

We have a good bunch of articles in this issue. Paul Chidley returns to give us an inside look at the YASBEC hardware. Jan Hofland, a new author for us, gives us the first part of building an "Arbitrary Waveform Generator," while Jay Sage continues his series on his custom designed home controller with a discussion of the hardware. Rounding out the hardware side: a brief on a hot new Z280 computer that Jay found in use during his trip to Germany last summer. Tilmann Rey tells us about it.

For you software junkies, Brad Rodriguez tells us how to write an assembler in Forth. Mind you, he isn't talking about writing *in* assembler—this article shows how to write the assembler itself. Great way to build your own cross assembler!

Al Hawley returns with his series on Z80 assembly language programming, while Terry Hazen picks up where Lindsay Haisley left off regarding IOPs. Add Matt Mercaldo's series on the F68HC11, Jay Sage's discussion of programming for compatibility, and the others, and we have a great issue for you!

By the way, a little "head's up." As you know, *TCJ* is published after-hours. Things will be a little rocky for the next few months. I am going on a string on business trips, back-to-back from now through December. Hang in with us, okay? In the meantime, issues 53 and possibly 54 may have to fall back to the 48 page format.

Should mention that the response to our reader survey has been outstanding. The responses were all very thought out and we will be using them to guide *TCJ*'s future. I plan to summarize after giving the stragglers a bit more time. If you haven't sent in your survey, please do so.

One comment came up several times: we need a "Letters-to-the-Editor" column. I agree.

Well, what is a an editorial column without an editorial? If you will, please join me for this month's diatribe:

When is the Public Domain not in the Public Interest?

This may verge on heresy, coming from one who edits the largest journal still supporting CP/M and runs a bulletin board dedicated to supplying public domain software, but I have a bone to pick. Stay with me just a bit.

See Editor, page 60

Reader-to-Reader

Recent GEnie messages, and issue 51, have given me much thought about *TCJ*, its popularity, and its "niche" in the marketplace.

What do I like about *TCJ?*

I've been getting *TCJ* since issue 43 or thereabouts. I originally subscribed because of its Forth coverage—being a Forth fanatic, I like to keep a complete library. But *TCJ* quickly "grabbed me" as much for its hardware-ish articles, and its coverage of CP/M.

TCJ has always been a "read ASAP" for me. I invariably start off by reading the editorial, *The Computer Corner* (which is usually relevant to my work), and *Real Computing* (which usually isn't, but which I enjoy reading—it helps me keep up with technology I don't normally use). Then, of course, the articles. I generally skip the Z-System articles, since I don't have Z-System and my CP/M systems have been down for a while—but I enjoyed the "history of Z-System" in the last issue, and thanks to *TCJ* I expect someday to be a Z-system user.

You summarized it best in your new back-cover ad, and in your latest editorial: I like articles that give me useful information, particularly on "how to do it" with limited or no resources. *TCJ* is a practical resource for me. I have already found many articles of professional value, and I expect to find many more.

What of the future of *TCJ?*

TCJ's target market seems to be serious hobbyists, novice Forthers, experienced CP/M users, and embedded systems developers—not that there's anything magical about embedded systems, except that they are the last bastion of small-scale engineering. But hobbyists at *TCJ*'s level are vanishing, and CP/M, while not dead, is certainly not burgeoning, either. Forthers and ESPers, of course, have competing journals.

In fact, I may be representative of this market...which is a problem for *TCJ*, since I'm a member of a dying breed! Low-budget hackers, who "cross-specialize" in both hardware and software, who like to tinker and know everything about how their computers work—these are being replaced by narrowly-focused specialists, "assembly line" programmers who don't know a parse tree from a pear tree, and board- swappers who don't know which end of the chip is pin 1. Sadly, that's where the money is these days.

How can *TCJ* grow?

Tom Peters (an author I highly rec-

See Reader, page 47

YASBEC

The Hardware

By Paul Chidley

Now that you were introduced to YASBEC (Yet Another Single Board Eight-bit Computer) in *TCJ* 51, I will try to put some more meat on the plate. Some of the information presented here is redundant but necessary if you are meeting Yas for the first time.

For several years I have been an avid 6502 enthusiast. It started with my first computer, an OSI Superboard, back in 1979. (my wife bought it for me) 1MHz 6502, 8K RAM, 4K Microsoft BASIC-in-ROM and some truly limited I/O. Not to mention the 300 baud Kansas City tape interface. Through the years the system grew and I grew with it. I have become a true expert on OSI computers and OS65D. However OSI made me a computer orphan in 1980 so recently I figured it was time to broaden my horizons. This year, I upgraded to CP/M. (I'll go to MesSyDOS when they work the bugs out). Problem was, I didn't have any hardware that would run CP/M. So like any good midnight hacker, I built some. Luckily I had a neighbour, Wayne Hortensius, who for years has been trying to convince me that I should switch to CP/M. His knowledge of CP/M and the Z-System help guide me in this project. The result of these efforts is YASBEC.

-Single Eurocard printed circuit board, double sided, soldermask.
-Z180 CPU, PLCC package.
-Static RAM, 32K to 1MB capacity.
-32Kx8 (27C256) EPROM.
-Optional Real-time clock, with battery backup (Dallas DS1216E).
-DD Floppy controller
-SCSI hard disk controller
-2 serial ports
-1 parallel printer port
-Buffered backplane connector
-Socketed for optional 9511A/8231A Arithmetic Processing Unit
-RAM and I/O addressing by PALs
-Requires +5VDC and +12VDC

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 on GEnie (email address: P.CHIDLEY), by regular mail at 162 Hunterhorn Drive NE, Calgary Alberta, Canada, T2K 6H5, or by phone at (403)274-8891 during reasonable MST hours.

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.

The board was designed on a single card Eurocard (100mm x 160mm) (approximately 4 x 6.5"). In other words, small. A Z181 with tsop RAMs and a 4 layer board would have resulted in a board half the size but at 5 times the price. Given the constraints of double sided and several large DIPs a large amount of circuitry was still squeezed onto the board, mostly thanks to a large number of surface mount components.

The Processor

The processor chosen was the Z180 or 64180Z. The Z80 is too limited and the Z181 and Z280 too expensive at 2.5 times the cost of a Z180. The Zilog Z180/Hitachi 64180Z processing unit contains a superset of Z80 instructions. There are 12 new instructions and many of the Z80 instructions execute in fewer clock cycles. The Z180 also contains a timing generator, two 16 bit timers, a clocked serial I/O port, a two channel DMA controller, two asynchronous serial ports and a memory management unit. There have been several articles in the past dealing with this chip should you require more information or programming details. For the YASBEC, a Z180 in a PLCC package saves space and provides A19, not available on the DIP package. This enables the Z180 to address 1MB of RAM without any external bank selects. The system is currently running with a clock speed of 9.216 MHz on the 10 MHz version Z180. Friends at Zilog have mentioned 12.5MHz, 16MHz, and even a 20MHz version some time down the road. Even the 20MHz chip is supposed to be available this year, but don't hold your breath. The 20MHz version would give us an 18.432 MHz system clock with approximately 55ns access times, but it remains to be seen if the YASBEC will be able to obtain these speeds reliably. The double sided design can be susceptible to noise and other problems at such speeds.

Memory

YASBEC's RAMs are static, provided by two 32 pin sockets. These sockets support 32Kx8, 128Kx8 or 512Kx8 RAMs in 600mil DIP packages. Static memory was chosen over dynamic for several reasons. Dynamic RAMs are cheaper but require several support chips taking up precious board space, they are not as fast as today's static, and they require CPU time for a refresh cycle. With the Z180 running a 9.216 MHz, access times of 100ns allow zero wait states with standard 'slow' SRAMs. Two 128Kx8 RAM chips provide a nice size for banked

CP/M at a moderate cost of around $30 each. The 128Kx8 chips are even available as fast as 25ns should you want to run the Z180 at 20MHZ with *no* wait states. It should however be noted that the super fast 128K chips are $100 but 6 months ago they were $300. By the time a 20MHz version of the Z180 is available these RAMs may be a more reasonable price. Using two 32kx8 chips requires a separate PAL to map the RAM in a contiguous block. While the Z180's memory management unit can map physical memory to 64k of logical memory the DMA controller uses 20 bit physical addresses. Therefor if you use DMA for I/O in the BIOS the DMA must know the physical location of the CP/M memory making it important that the memory be in one block.

Monitor

The system includes a monitor ROM program available at power up if no drives and/or software are found or if a key is pressed before disk bootup. The ROM is a 27C256 providing 32Kx8 of space. The first version of the monitor occupied 4K. There is room for expansion. Several people have asked why such a big EPROM for so little code. This however is a common problem today especially in embedded controller applications. The answer; no one makes small, slow EPROMs anymore. A 27C256 is cheaper than a 27C64. Of course with the MMU (memory management unit) in the Z180 this is really a non-issue. Once we boot we swap out the EPROM, anyway. Using the Dallas DS1216E SmartWatch provides a battery backed-up clock for the system. The SmartWatch also serves as a socket for the EPROM. While the optional SmartWatch may seem pricey at $25, anyone that has to enter the date and time more than twice will soon want one.

Floppy Disk

The floppy disk schematic is shown in *figure* 1. The controller is a Western Digital WD-1772-02 also available from VTI as the VL-1772-02. The 1772 (U14) includes everything needed for floppy control in a 28 pin DIP except buffers and an 8MHz clock. The 8MHz clock comes from the oscillator module (U24). The 4MHz and 2MHz signals from U23 are used elsewhere by the APU (arithmetic processing unit) and not by the floppy circuit. A 74AC04 (U22) provides the buffering while a 74AC00 (U18) gates motor_on with the drive select lines. The four drive select lines come from the control register (U20) in *figure* 2. Note that _FRESET and _DDEN also come from the control register. _DDEN selects single or double density while _FRESET gives us software control of the 1772 reset line. This allows the software to force a reset should the chip become hung waiting for a condition too long. e.g. timeout waiting for a floppy that isn't there. The major disadvantage of the 1772 is that it does not support quad-density (i.e. 1.44mb) drives or 8" drives.

Table 1.

```
A Description of the 128MEM1 pal.        PAL16L8

;-------------------------------------- PIN Declarations -------------

PIN  1        A19            COMBINATORIAL         ; INPUT
PIN  2        A18            COMBINATORIAL         ; INPUT
PIN  3        A17            COMBINATORIAL         ; INPUT
PIN  4        A16            COMBINATORIAL         ; INPUT
PIN  5        A15            COMBINATORIAL         ; INPUT
PIN  6        RD             COMBINATORIAL         ; INPUT
PIN  7        WR             COMBINATORIAL         ; INPUT
PIN  8        MREQ           COMBINATORIAL         ; INPUT
PIN  9        IORQ           COMBINATORIAL         ; INPUT
PIN  10       GND                                  ;
PIN  11       IO_A7          COMBINATORIAL         ; INPUT
PIN  12       EPROMCS        COMBINATORIAL         ; OUTPUT
PIN  13       SRAM1CS        COMBINATORIAL         ; OUTPUT
PIN  14       SRAM2CS        COMBINATORIAL         ; OUTPUT
PIN  15       MEMRD          COMBINATORIAL         ; OUTPUT
PIN  16       MEMWR          COMBINATORIAL         ; OUTPUT
PIN  17       IORD           COMBINATORIAL         ; OUTPUT
PIN  18       IOWR           COMBINATORIAL         ; OUTPUT
PIN  19       BUS_DIR        COMBINATORIAL         ; OUTPUT
PIN  20       VCC                                  ;

;------------------------------------ Boolean Equation Segment ------

EQUATIONS
/EPROMCS = /MREQ * /A19 * /A18 * /A17 * /A16 * /A15
/SRAM1CS = /A19 * /A18 * A17
/SRAM2CS = /A19 * A18 * /A17
/MEMRD = /RD * /MREQ
/MEMWR = /WR * /MREQ
/IORD =  /RD * /IORQ
/IOWR =  /WR * /IORQ
/BUS_DIR = /( A19 * /MREQ * /RD + IO_A7 * /IORQ * /RD )

;--------------------------------------------------------------------

/EPROMCS = $0:0000 - $0:7FFF    32k x 8 Eprom (27C256)
/SRAM1CS = $2:0000 - $3:FFFF    128k x 8 Static Ram
/SRAM2CS = $4:0000 - $5:FFFF    128k x 8 Static Ram
/BUS_DIR = Reverse data bus buffer for a
           memory read from $8:0000 - $F:FFFF
           or an i/o read from $80 - $FF

$F:FFFF  _______________
        |               |
        |               |
$6:0000 |_______________|
        |               |
        |               |
        |               |
$5:0000 |               |
        | Static Ram    |
        | 128K x 8      |
        | TC551001PL-100|
$4:0000 |_______________|
        |               |
        |               |
        |               |
$3:0000 |               |
        | Static Ram    |
        | 128K x 8      |
        | TC551001PL-100|
$2:0000 |_______________|
        |               |
        |               |
        |               |
$1:0000 |_______________|
        |               |
        |_______________|
$0:7FFF | Monitor Rom   |
$0:0000 |_32k_x_8__27C256_|
```

SCSI

A DP5380V or DP8490V (U21) SCSI controller is available on the board for hard disks as shown in *figure 3*. The hardware is capable of supporting up to 7 devices on this one bus. For those of you not familiar with SCSI (shame on you) all you need is a SCSI drive such as an ST157N and a 50 pin ribbon cable and you have a hard disk drive on your system. Of course you still need some software to talk to it. RS1, RS2 and RS3 provide the standard termination required at each end of a SCSI bus. Another set must be provided on the other 'end' of the bus (i.e. the last drive in the chain). The SCSI unit number of the YASBEC is coded in software to seven, providing jumpers to make this address selectable was not as important as saving board space. The only reason anyone would need a different unit number would be to put two (or more) YASBECs on the same SCSI cable and should anyone really want to do that a simple change to the BIOS can give one of the boards a new address. Interfacing to the chip is done with _NCRCS, _IORD, _IOWR and _RESET. The other four lines provide feedback for interrupts and DMA control. The actual controller is built into the drive, the 5380 is actually just a fancy parallel port to talk to the smarts on the drive via the SCSI bus. The SCSI transfer protocol is mostly handled in software. The 5380 is capable of polled I/O, pseudo DMA and full DMA data transfers. It is common to find 20 MB SCSI drives for $200 in *Computer Shopper* and other such magazines. At those prices it is hardly worth running an old flea market special with half its heads missing, even if it was $20. (hi, Hal!)

Table 2.

```
A Description of the YASIO-1 pal.       PAL16L8

;------------------------- PIN Declarations ------------

PIN  1        A7                 COMBINATORIAL ; INPUT
PIN  2        A6                 COMBINATORIAL ; INPUT
PIN  3        A5                 COMBINATORIAL ; INPUT
PIN  4        A4                 COMBINATORIAL ; INPUT
PIN  5        A3                 COMBINATORIAL ; INPUT
PIN  6        A2                 COMBINATORIAL ; INPUT
PIN  7        A1                 COMBINATORIAL ; INPUT
PIN  8        A0                 COMBINATORIAL ; INPUT
PIN  9        RD                 COMBINATORIAL ; INPUT
PIN  10       GND                              ; INPUT
PIN  11       WR                 COMBINATORIAL ; INPUT
PIN  12       IO_A7              COMBINATORIAL ; OUTPUT
PIN  13       IORQ               COMBINATORIAL ; INPUT
PIN  14       FPPCS              COMBINATORIAL ; OUTPUT
PIN  15       NCRCS              COMBINATORIAL ; OUTPUT
PIN  16       NCRDACK            COMBINATORIAL ; OUTPUT
PIN  17       FDCCS              COMBINATORIAL ; OUTPUT
PIN  18       PRINTERSTROBE      COMBINATORIAL ; OUTPUT
PIN  19       CREQCS             COMBINATORIAL ; OUTPUT
PIN  20       VCC                              ; INPUT

;------------------------ Boolean Equation Segment ------

EQUATIONS
/CREQCS = /IORQ * /WR * /A7 * A6 * /A5 * /A4 * /A3 * /A2 * /A1 * /A0
/PRINTERSTROBE = /IORQ * /WR * /A7 * A6 * /A5 */A4 */A3 */A2 * A1
/FPPCS = /A7 * A6 * /A5 * A4 * /A3 * /A2 * /A1
/NCRCS = /IORQ * /A7 * A6 * /A5 * A4 * A3
/NCRDACK = /IORQ * /A7 * A6 * A5 * /A4 * /A3 * /A2 * /A1 * /A0
/FDCCS = /IORQ * /WR * /A7 * A6 * A5 * /A4 * A3
       + /IORQ * /RD * /A7 * A6 * A5 * /A4 * A3
/IO_A7 = /A7

;--------------------------------------------------------
                   $00 - $3F // Z180 Internal I/O
        /CREQCS = $40       // Control Register (74hc273)
/PRINTERSTROBE = $42 - $43 // Printer Strobe
         /FPPCS = $50 - $51 // Floating Point Processor
         /NCRCS = $58 - $5F // SCSI
       /NCRDACK = $60       // SCSI DACK
         /FDCCS = $68 - $6F // Floppy Disk Controller
                  $70 - $7F // Not Used
                  $80 - $FF // Off board I/O


   $FF   ___________________
        | Off board I/O     |
   $80  |___________________|
        | Not Used          |
   $6F  |___________________|
        | Floppy Controller |
   $68  |___________________|
        | SCSI DACK         |
   $60  |___________________|
        | SCSI Controller   |
   $58  |___________________|
        | Arith. Math. Proc |
   $50  |___________________|
        | Printer Strobe    |
   $42  |___________________|
        | Control Register  |
   $40  |___________________|
        | Z180 Internal I/O |
   $00  |___________________|
```

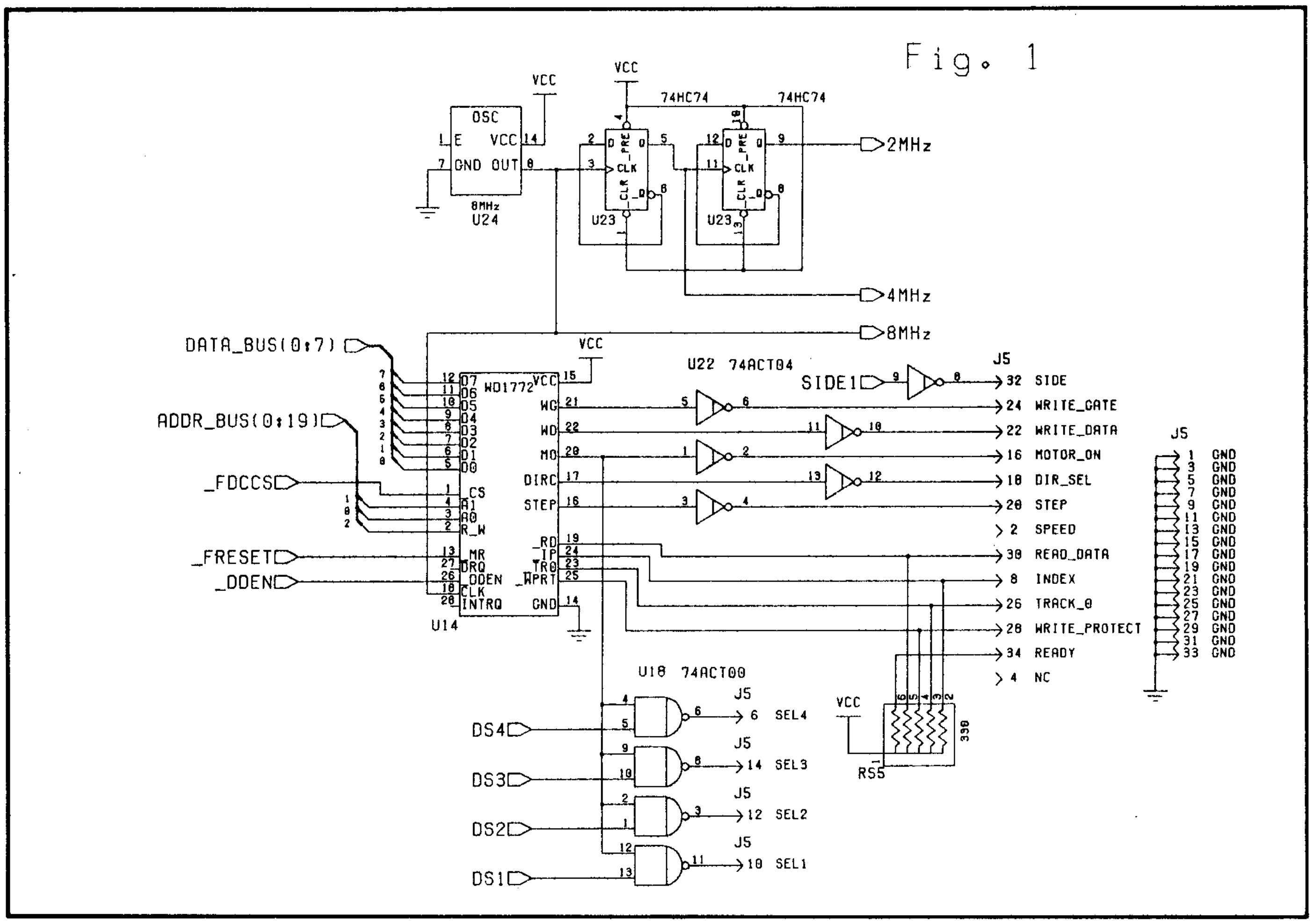
Fast Math

YASBEC is a blend of very new parts and some very old. The very old part is the AMD 9511A/Intel 8231A Arithmetic Processing Unit. While this chip has been around since the early days of computers it is still in current data books. Why? No one has ever built anything to replace it. Current co-processors are just that, "co"-processors. They are designed to work with their mate but not any others. The APU (9511A/8231A) on the other hand is easily interfaced with most common 8 bit processors. It features fixed point single and double precision (16/32 bit), floating point precision (32 bit), add, subtract, multiply, divide, conversions and floating point trigonometric, inverse trigonometric, square roots, logarithms and stack oriented operand storage. The down side is that it runs very hot, and costs $150! Almost as much as the rest of the board. So why was one included in the design? For fast math and Wayne had one. The APU is of course optional, YASBEC will do just nicely without one installed. But if you need to do sin(x) a thousand times and money is no object then there is a socket for an APU. Wayne has an astronomy program (sky32) that number crunches then plots a graphics display four times faster with the APU than without. If nothing else it might be worth the money just to show you neighbour how fast your homebrew can crunch numbers compared to his PC.

Address PALs

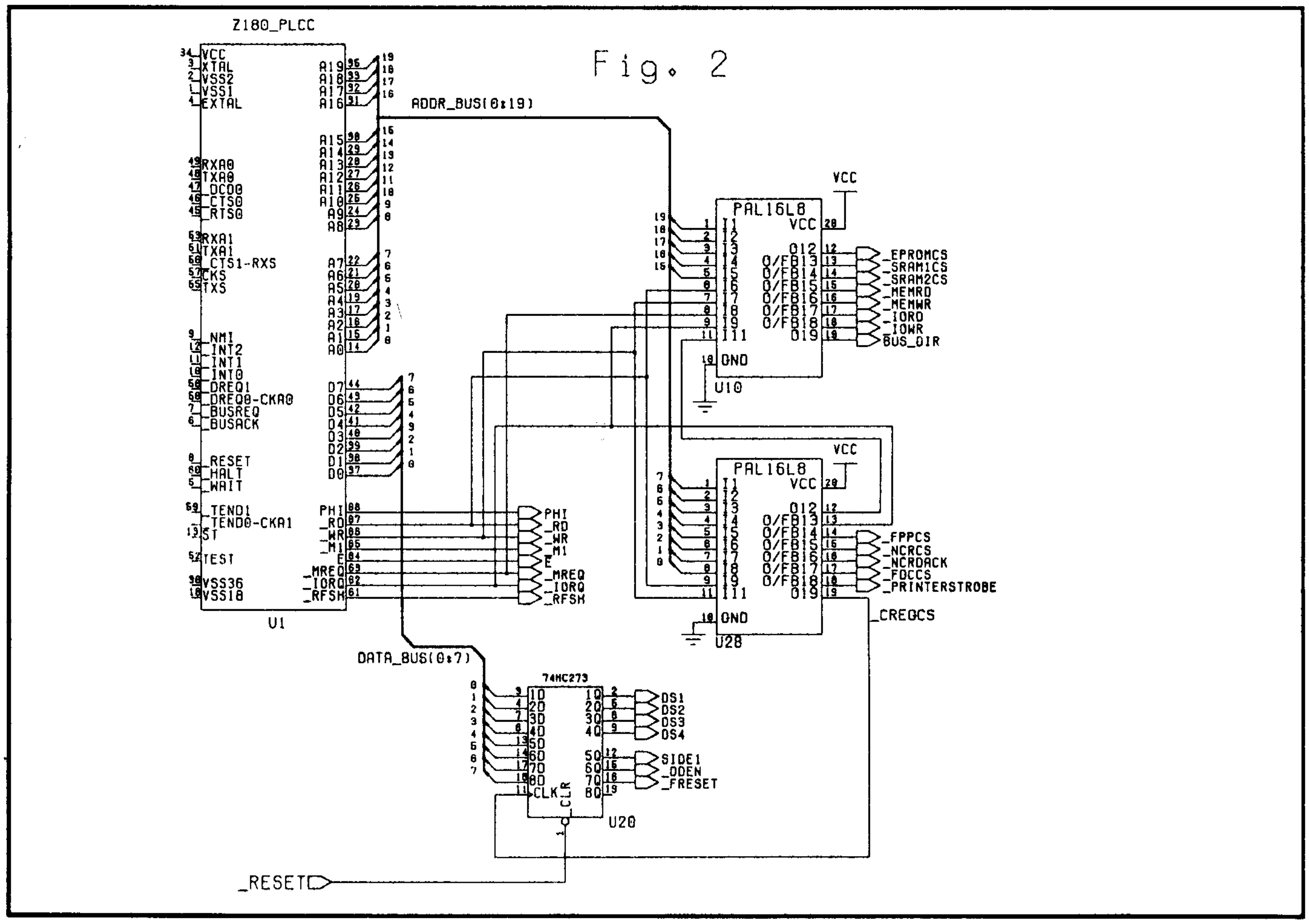
Memory and address decoding is done with two 16L8 PALs, see *figure* 2. The 16L8 is easy to program and its age makes the older bipolar versions available for as little as $2 with 15ns delays. Should it be needed the 16L8 (16V8) is also available as fast as 5ns but at a price. As you can see, PALs make the addressing schematic simple, but to see what is going on you have to know what the PAL does. *Table 1* is a listing of the PAL equations and memory map. Given the pin definitions and the PAL equations you can see how this PAL decodes memory to fit the desired memory map. Having all the memory address decoding in one PAL improves the access time needed to talk to the RAM. A 15ns PAL with 100ns RAM legally meets the worst case op-code fetch of 118ns at 9.216 MHz. Where the RAM is located in the 1MB map was

Fig. 1



OSC
8MHz
U24
VCC
74HC74
U23
2MHz
4MHz
8MHz
DATA_BUS(0:7)
ADDR_BUS(0:19)
_FDCCS
_FRESET
_DDEN
WD1772
U14
U22 74ACT04
SIDE1
J5
32 SIDE
24 WRITE_GATE
22 WRITE_DATA
16 MOTOR_ON
18 DIR_SEL
20 STEP
2 SPEED
30 READ_DATA
8 INDEX
26 TRACK_0
28 WRITE_PROTECT
34 READY
4 NC
GND
U18 74ACT00
DS4
DS3
DS2
DS1
6 SEL4
14 SEL3
12 SEL2
10 SEL1
RS5
330

Fig. 2



Z180_PLCC
U1
ADDR_BUS(0:19)
PAL16L8
U10
_EPROMCS
_SRAM1CS
_SRAM2CS
_MEMRD
_MEMWR
_IORD
_IOWR
BUS_DIR
PAL16L8
U28
_FPPCS
_NCRCS
_NCRDACK
_FDCCS
_PRINTERSTROBE
_CREGCS
PHI
_RD
_WR
_M1
E
_MREQ
_IORQ
_RFSH
DATA_BUS(0:7)
74HC273
U20
DS1
DS2
DS3
DS4
SIDE1
_DDEN
_FRESET
_RESET

and still is an issue. In the PAL currently being used the memory starts at $2:0000 leaving a hole between it and the EPROM. Not a big deal, unless an external memory board is used and people want the full 1MB (less 32K EPROM) for RAM. The joys of a PAL, should it come to that, is we just burn a new one. The current memory PAL (U10) also controls the data bus buffer direction. To do this it needs to know what RAM is on the YASBEC and what is external. For now we just chose $8:0000 and up as external but this too can change. The I/O PAL can also reverse the external data bus buffer. The I/O PAL is set up for the Z180 internal I/O from $00-$3F, the YASBEC on-board I/O from $40-$7F, and off-board I/O from $80-$FF. The I/O PAL contents are listed in *Table* 2.

Serial Ports

YASBEC is a serial stand-alone system. To get a working system you need to talk to the board. In other words, it is a complete stand-alone system but you need a dumb serial terminal to talk to it. An AT works quite nicely. In fact, many CP/Mers I have talked to say that a dumb terminal is a very suitable application for an AT. A MAX239 provides TTL to RS-232 conversion for all data and signal lines available from the Z180's serial ports as illustrated in *figure 4*. The MAX239 is from the same family as Maxim's MAX232 but with 3 transmit and 5 receive lines instead of 2 and 2. Unlike the MAX232 which only requires +5V the MAX239 requires +5V and +12V. The +12V should be readily available since it is also needed for any disk drives. The console port has a single flow control line labeled RTS coming from the terminal (DCE). This definition was taken from the Z180 and is intended as a hardware handshaking line. The problem is that most terminals either don't have hardware handshaking or use some other pin for this other than RTS. How you would actually use this line, if at all, depends on your terminal. The modem port has three control lines, DSR, DTR and DCD. The only gotch-ya here is that DCD actually disables the serial port hardware when not true. So unless you have a modem with a true RS232 conversation mode you will need to set jumpers U29 and U30. Some modems have DCD true while on-hook in conversation mode, they then set it false after a dial command after which it represents the state of the carrier. With such a modem you may hook DCD up as originally intended. U30 allows you to use DSR instead while U29 allows you to short _DCD to ground making it always true. Both ports do an adequate job despite the limitations presented by the Z180's architecture. Note the reset circuit in figure 4 has three inputs. C3 and R4 provide a power up reset, shorting S2 provides a reset as well as shorting pins 9 and 10 on J3 the console serial port.

Printer Port

The Centronics compatible parallel printer port is your standard eight bit output port with hardware handshaking as shown in *figure 5*. Writing to _printerstrobe+1 latches the data in U16, sets _strobe low via address line 0 and U15, and clears _int2 if set. Writing to _printerstrobe sets _strobe high, the printer acknowledges that the data was received with _ack through U15 which sets _int2 low until the next _printerstrobe.

See YASBEC, page 42

Fig. 3

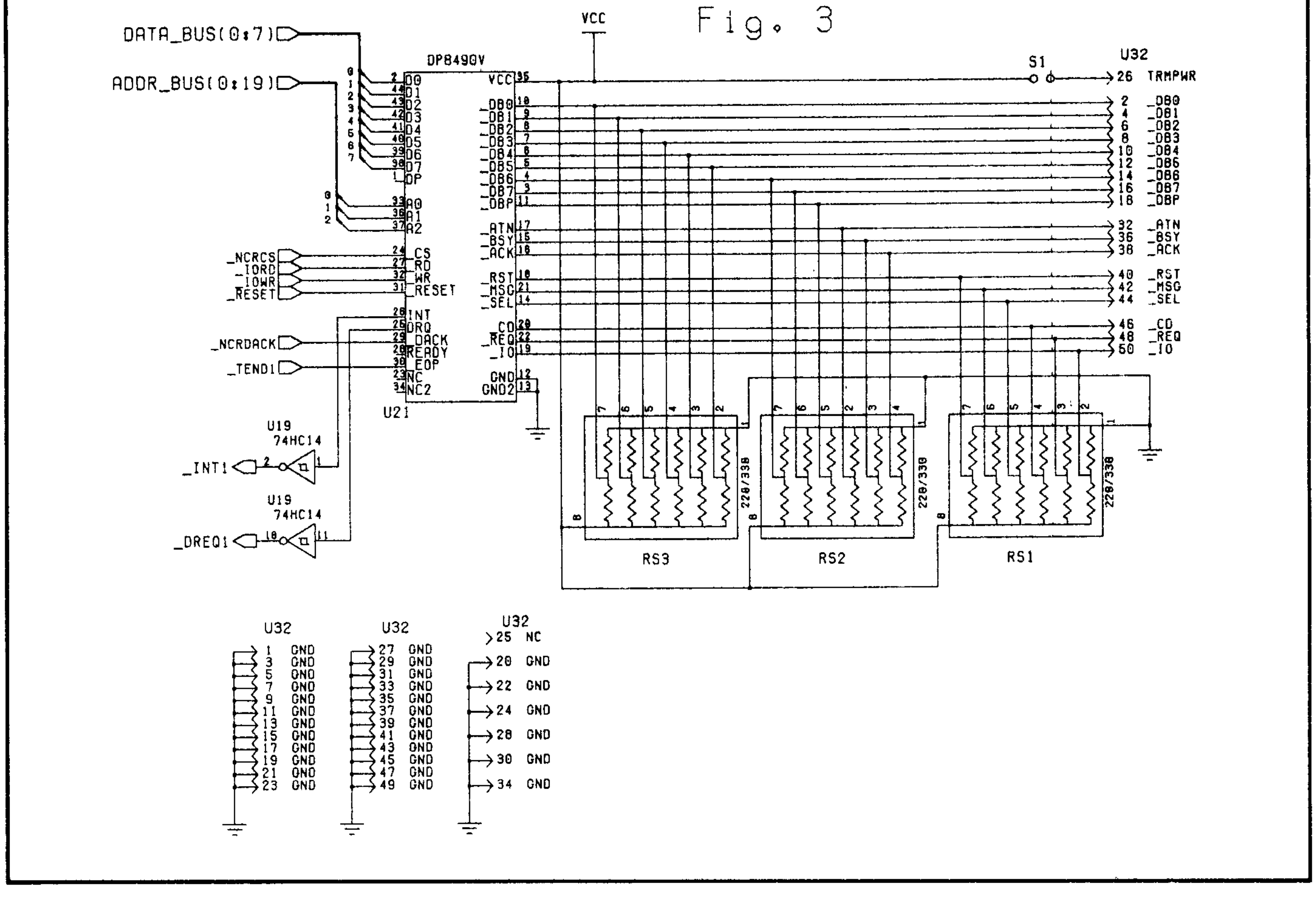


Fig. 4

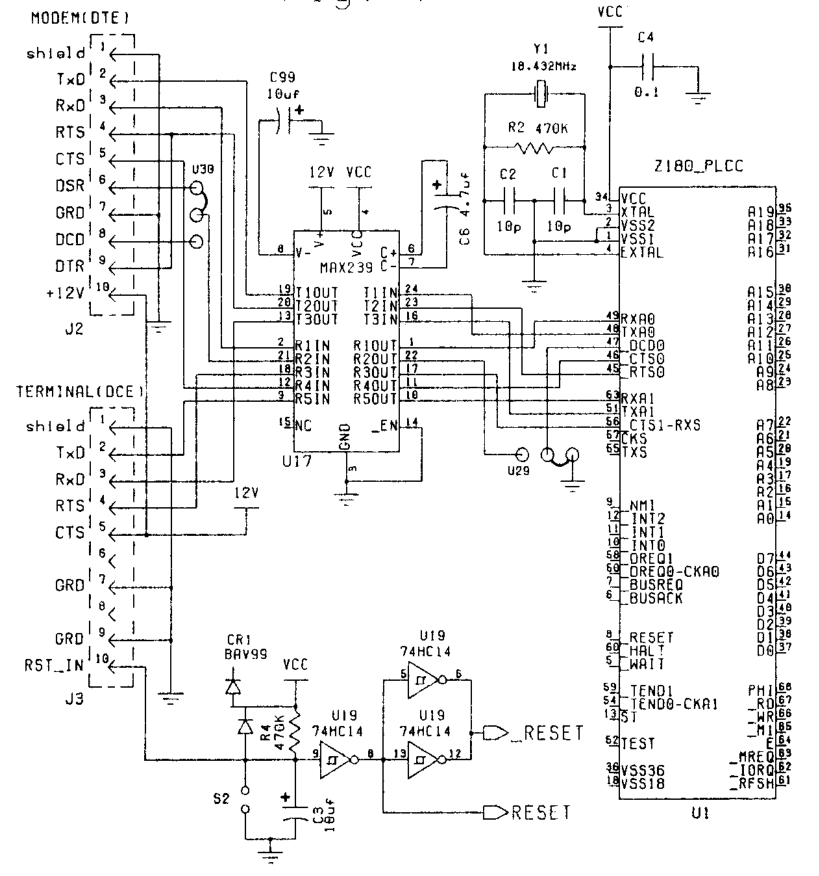
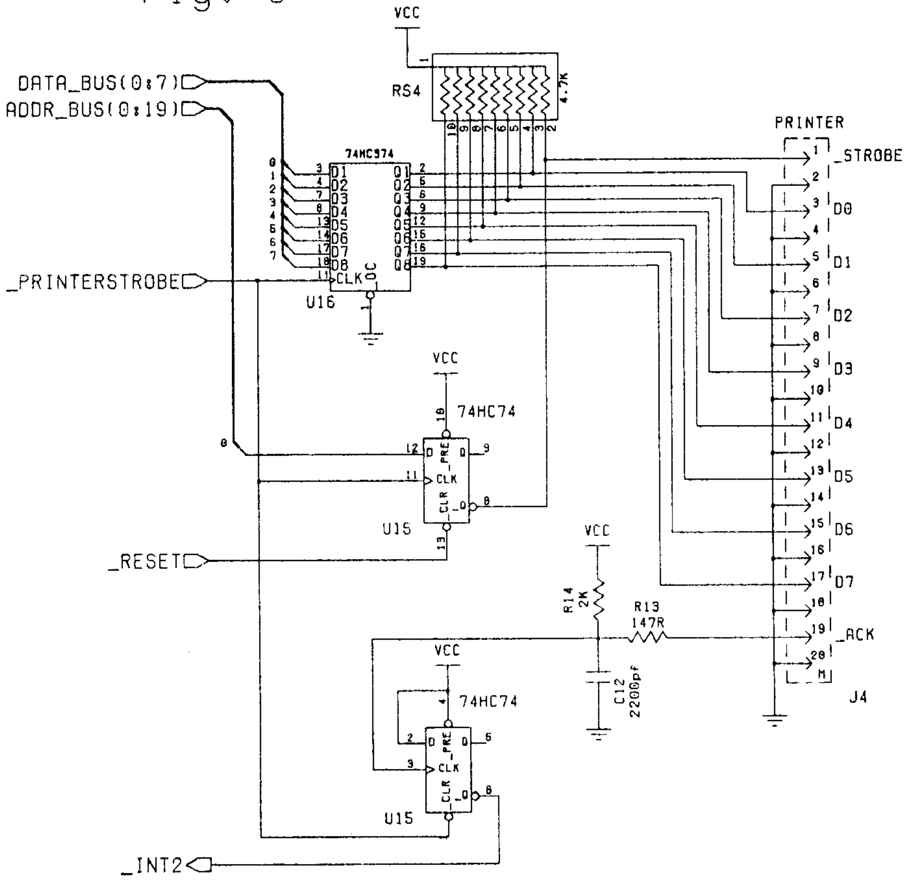

MODEM(DTE)
shield
TxD
RxD
RTS
CTS
DSR
GRD
DCD
DTR
+12V
J2
TERMINAL(DCE)
RST_IN
J3
U30
C99 10uf
12V VCC
MAX239
U17
C6 4.7uf
Y1 18.432MHz
R2 470K
C2 10p
C1 10p
VCC
C4 0.1
Z180_PLCC
U1
U29
CR1 BAV99
R4 470K
U19 74HC14
S2
C3 10uf
_RESET
RESET

Fig. 5


DATA_BUS(0:7)
ADDR_BUS(0:19)
_PRINTERSTROBE
_RESET
_INT2
RS4 4.7K
74HC374
U16
74HC74
U15
VCC
R14 2K
R13 147R
C12 2200pf
PRINTER
_STROBE
D0
D1
D2
D3
D4
D5
D6
D7
_ACK
J4

An Arbitrary Waveform Generator

Using the Harris RTX2001A

By Jan Hofland

Introduction

Vibration control systems for mechanical structural testing generally require a programmable signal source. This source is used to control the system stimulus during test, which is typically a hydraulic shaker table. The usual test setup consists of a stimulus and several response measurement channels. The mechanical motion of the unit under test is measured with many accelerometers attached to it. The output from the accelerometers goes to an analog to digital converter and then through digital signal processing for extraction of the pertinent data to characterize the mechanical modes of vibration and sensitivities to various stimulii. An important attribute of the signal source used in this type of testing is the ability to be programmed in real time during the testing so that the stimulus can be adapted to the measured responses.

The purpose of this design project is to demonstrate the capability of a programmable waveform generator designed around the Harris RTX2001A microprocessor, a Digital-to-Analog Converter (DAC), and a programmable lowpass filter. The RTX2001A is used for controlling the DAC sample rate (when the next output value is put into the DAC), for inputting the data to be output, for putting that data to be output into one of two data queues, for controlling the data queues and extracting data from the queues to be output to the DAC, and for controlling the output lowpass filter cutoff. It is also used to generate sinewave data of any frequency up to 40 Khz.

The two data queues are implemented in software. They're controlled with a write pointer for inserting values, a read pointer for extracting values, an end pointer for determining when to loop back to the start of the queue, and a count variable to keep track of the number of values in the queue. They can be used as a circular buffer where some periodic waveform is loaded into the buffer and then the data is output from the buffer repeatedly to generate a continuous output. The buffers can be configured to operate in a ping pong fashion, where data is extracted from one queue until it is empty and then it automatically switches to the other queue and extracts data from it until it empties, and then it stops. Another operating mode is to extract data from one buffer until it is empty.

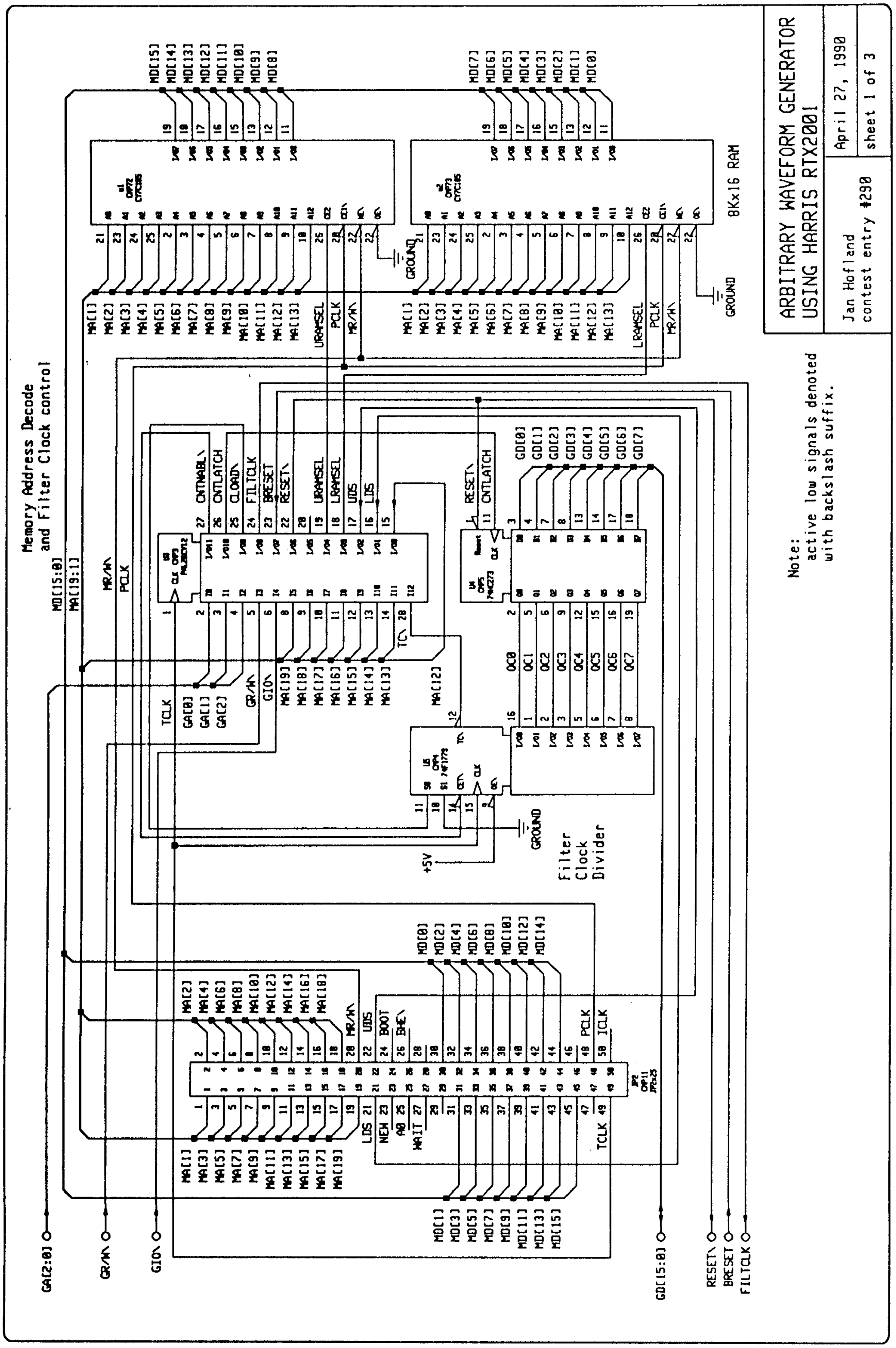
Sinewave outputs are generated by using a fixed output sample rate, a phase accumulator, table lookup of successive values, and interpolation between values in the table. There are two basic ways of generating sine values of varying frequency. One method uses a fixed number of points per cycle of the sinewave and varies the rate at which these samples are output. The other method is to fix the output sample rate and to vary the number of points output per sinewave cycle. In other words, vary the phase increment between successive outputs. This is the method of sinewave generation used here. There is an 800 point sine table representing one full cycle of the sinewave. The phase accumulator is incremented by a value dependent on output frequency each time a value is output to the DAC to determine the next value to be output. There are two sinewave output modes, one optimized for speed and one optimized for accuracy and versatility. In the high speed mode, a fixed amplitude sinewave is output using values directly from the sinewave table. In the 'accusine' mode there is a second phase accumulator for keeping track of the intermediate phase between table points. The value output is calculated using the sum of two angles trigonometric relationships. This mode also allows amplitude scaling and D.C. offset adjustment.

Hardware Overview

The block diagram for the hardware added to the Harris RTX2001A Evaluation Board is shown as Figure 1. It consists of the following blocks:

- An additional 8K x 16 of RAM. Implemented with two CY7C185 static RAMs, 45 ns access time.
- Memory Address Decoder. Implemented in part of a GAL 26CV12, 15 ns device. Maps the added RAM into memory space 05000-08FFE, hex.
- Input FIFO, 2K x 16. Implemented with two CY7C429 FIFOs, 40 ns access time. Used for command and data input.
- FIFO Read/Write Control. Interlocks the reads and writes with their respective empty and full flags. Provides write pulsewidth control and interfaces to the ASIC bus for reading the FIFO. Implemented in part of a GAL 22CV10, 15 ns device.
- Filter Clock Programmable Counter. Used to control the toggle rate of the switched capacitor lowpass filter for wide bandwidths. Implemented using a 74F1779 counter and 74HC273 register.
- Filter Clock Control. Controls filter counter reload, counter enable, and writing to the counter reload register. Interfaces to the ASIC bus. Implemented with part of the GAL 26CV12 device.
- DAC Data Shift Register. Pro-

Jan Hofland is employed with Hewlett Packard as a hardware design engineer, working primarily on 68K based systems. His personal interests include woodworking, and playing with electronics for over 20 years. His current favorite system is the F68HC11 from New Micros. Jan can be contacted at 2419 123rd Ave. SE, Everett WA 98205 or by telephone (206) 334-0738 during the evenings.



ARBITRARY WAVEFORM GENERATOR
USING HARRIS RTX2001
Jan Hofland
contest entry #290
April 27, 1990
sheet 1 of 3
Memory Address Decode
and Filter Clock control
8Kx16 RAM
Filter
Clock
Divider
Note:
active low signals denoted
with backslash suffix.
GROUND
+5V
MD[15:0]
MA[19:1]
MR/W\
PCLK
TCLK
GA[2:0]
GR/W\
GIO\
GD[15:0]
RESET\
BRESET
FILTCLK
CNTNABL\
CNTLATCH
CLOAD\
URAMSEL
LRAMSEL
UDS
LDS
BOOT
BHE\
NEW
AB
WAIT
ICLK

vides parallel to serial data conversion for the output DAC. Data is input from the ASIC bus. Implemented using two 74HC299 devices.

• Serial DAC Data Control. Controls data loading and shifting and when data is latched into the DAC. Implemented using part of the GAL 22CV10.

• Serial Input DAC. An 18 bit DAC. This device is a Burr-Brown PCM61P.

• Switched Capacitor Lowpass Filter. Used for output waveform reconstruction. Tuned to a particular cutoff by varying the clock frequency. This device is a Linear Technology LTC1064-1, 8 pole, elliptic function lowpass filter.

• Output Buffer. Unity gain. Implemented using an LM6321.

The schematics are included as *Figure* 2. The ABEL source code for the GAL 26CV12, U3, and GAL 22CV10, U8, will be printed in a future issue of *TCJ*.

Hardware Detailed Description

External Input Port

Commands and data are input to the Waveform Generator via the FIFO, comprised of two Cypress Semiconductor CY7C429 devices, U9 and U10. The FIFO provides a 16 bit parallel interface by 2048 words deep. Inputs are pulled up to +5V via 4.7K resistors. Data is read from the FIFO by the RTX2001 via the ASIC bus. The FIFO includes empty and full flag outputs.

FIFO reads and writes are controlled by circuitry in the 22CV10 PLD, U8. Data is read from the ASIC bus at address 19 hex. FIFORD is asserted active low if the ASIC bus address GA[2:0]=1, GIO is asserted, GR/W is high, TCLK is low and the FIFOMT flag is not asserted. Once asserted, FIFORD stays asserted until TCLK goes high again. This is to prevent short cycling when reading the last valid value from the FIFO and the FIFOMT flag is asserted. The PLD used is a 15 ns device, so FIFORD is asserted 15 ns after TCLK goes low, worst case. The CY7C429 FIFO has a 40 ns access time, worst case, so the FIFO data will be valid on the ASIC bus at least 7 ns before cycle completion. In actual operation, typical access times of less than 30 ns are common. The reason for using TCLK instead of PCLK is that TCLK is being used in the PLD for other reasons and that for ASIC bus operations, TCLK and PCLK have the same timings.

FIFO write operations are controlled by the active low STROBE input and a flip-flop in PLD U8. When STROBE is sensed active low by U8 and if the FIFOFUL flag is not asserted, then U8 sets FIFOWR active low. When the external interface senses that FIFOWR is active, it then may release STROBE. The data must be held valid by the external interface until FIFOWR is deasserted. When U8 senses that STROBE is deasserted, it deasserts FIFOWRand strobes the data into the FIFO. This interlock protocol ensures that only one value is written into the FIFO for each STROBE pulse and that the write pulsewidths are sufficient for the FIFO. The maximum transfer rate into the FIFO is limited to one-half the system clock rate.

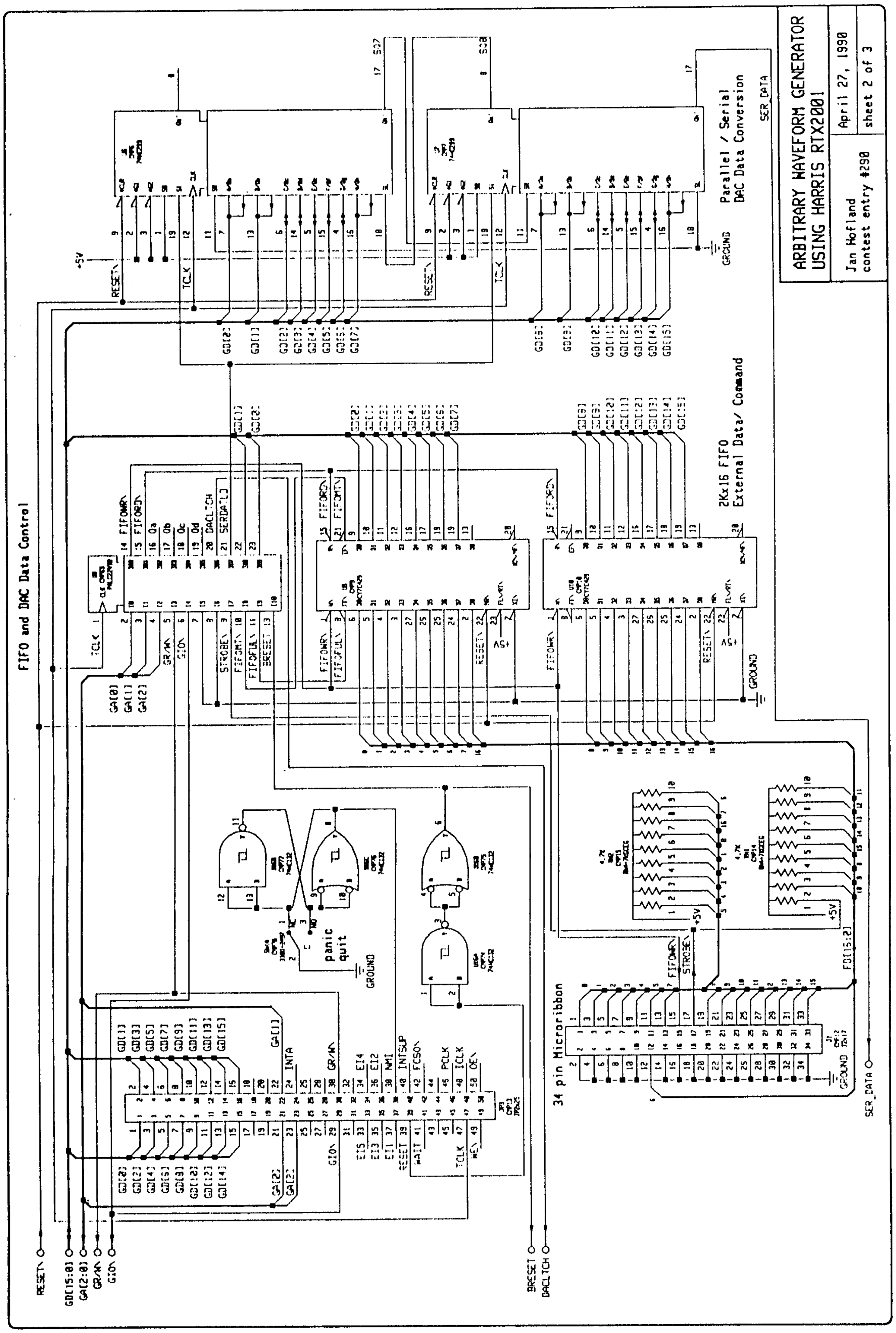
Listing 1.

```
The following 16 specific commands have been implemented:

   command          description

 0 NOP              no operation
 1 fastSine         set up to operate in fast sine mode.  The next value
                    input is the frequency.
 2 accuSine         set up to operate in the accurate sine mode.  The next
                    value input is the frequency.
 3 setAmpl          set output amplitude, in millivolts, for accurate sine
                    output mode.  The value following this command is used
                    as the output amplitude. This value will be truncated
                    to a multiple of 50 mV.
 4 bufLoop0         set up to output from buffer 0 in circular buffer mode
 5 bufLoop1         set up to output from buffer 1 in circular buffer mode
 6 pingPong0        set up to output in ping pong mode starting with
                    buffer 0
 7 pingPong1        set up to output in ping pong mode starting with
                    buffer 1
 8 bufOut0          set up to output from buffer 0 until it is empty
 9 bufOut1          set up to output from buffer 1 until it is empty
10 ldBuf            used to select a buffer and load it with data values.
                    The next parameter must be the number of values to be
                    loaded, up to 2048, and the msb set if it is for
                    buffer 1.  Then the data values are input in sequence.
11 setOffset        set the offset to be subtracted from the sinewave
                    value in accurate sinewave mode, in 100's of microvolts
                    The next value following this command is the offset.
12 setFilt          used to set the filter cutoff point.  The next value
                    input is used for the filter cutoff, range 1 to 40000
                    Hz.
13 getPeriod        used to input the desired sample rate.  The next
                    value input is used for the timer 0 period.  If in
                    either sinewave operating mode, a check is performed
                    to readjust the phase increment to be consistent with
                    this period and the output frequency.
14 stop             disable timer 0 interrupt to stop outputting points
15 start            enable the timer 0 interrupts to start outputting data
                    and then go into a loop to repeatedly check the stale
                    data flag in register RX and update the value in
                    scratchpad RH if the stale data flag is set.  Continue
                    this loop until the timer 0 interrupt is masked.

The execution vectors for each of the 16 commands are contained in a table
```

Digital to Analog Converter

The DAC chosen for this application is a Burr-Brown PCM61P, U14. It was chosen primarily for its low cost, at 16 bit capability, and compact size.

The PCM61P is a serial input 18 bit device and includes an integral amplifier for current-to-voltage output conversion. The only digital inputs required are the serial data input, a clock, and a latch enable signal. It accepts data most significant bit first, binary 2's complement coded. Data is clocked into the device on the rising edge of the clock. The maximum clock rate is 16.9 MHz, well beyond our 8 MHz system clock rate. The latch enable signal actually controls which information is transferred to the digital-to-analog circuitry from the input interface in the part. The input clock and the serial data stream may be thought of as a continuous input. The last 18 bits input before the latch enable makes a high-to-low transition are transferred to the DAC conversion circuitry in the PCM61P. The generation of latch en-


ARBITRARY WAVEFORM GENERATOR
USING HARRIS RTX2001
Jan Hofland
contest entry #290
April 27, 1990
sheet 2 of 3
FIFO and DAC Data Control
2Kx16 FIFO
External Data/ Command
Parallel / Serial
DAC Data Conversion
SER_DATA
34 pin Microribbon
panic
quit
GROUND
+5V
RESET\
GD[15:0]
GA[2:0]
GR/W\
GIO\
BRESET
DACLTCH
FD[15:2]
TCLK

able is controlled by PLD U8.

The latch enable signal, DACLTCH, is generated by the circuitry in PLD U8 based on a 16 bit up/down gray scale counter. The counter state variables are Qd, Qc, Qb, and Qa. The counter normally is in idle state 0. When a 16 bit value is loaded into the DAC data shift register (U6 and U7), it enables the counter to start counting up in the sequence 0, 1, 3, 2, 6, ... to the end state 8. The DACLTCH goes high. Now the gray code counter reverses direction and starts counting down through the states 9, 11, and 10. DACLTCH stays high during these states. When DACLTCH is high, the successor state of state 10 is the idle state, state 0. When this state transition occurs, DACLTCH is held high for the high half of TCLK and then deasserted. Thus DACLTCH makes its high-to-low transition one half clock cycle after 18 data bits have been clocked into the DAC.

The DAC output is a voltage ranging from -3V to +3V. This output is connected directly to the switched capacitor lowpass filter, U15.

DAC Shift Register

Two 74HC299 shift registers, U6 and U7, are used to load serial data into the DAC. The shift registers are clocked by TCLK, the same as the DAC. Normally, the shift registers are configured to shift data left, shifting data out of the most significant bit (Qh' of U7) to the DAC and shifting zeroes in (SR input of U6). New data is loaded into the shift register from the ASIC data bus, GD[15:0] by writing to ASIC address 18 (hex). The address is decoded by PLD U8 to assert SERDATLD to the shift register. This write also initiates the DAC latch enable control, already described.

Programmable Lowpass Filter

The output from the DAC is filtered by a switched capacitor lowpass filter, U15. This device is an LTC1064-1 made by Linear Technology Corporation. It is an 8th order, clock tunable, elliptic function (Cauer) lowpass filter with +/- 0.1 dB passband ripple and 72 dB stopband attenuation at 1.5 times the cutoff frequency. The cutoff frequency is 1/100th of the clock input frequency, so the lowpass setpoint is easily controlled by changing the clock frequency. All this in only one 14 pin DIP package! The output from the lowpass filter is buffered by a unity gain buffer, U13, to drive the output.

Filter Clock Control

There are two mechanisms used to control the filter clock frequency. For wide bandwidths, the filter clock is controlled by an 8 bit programmable down counter, U5 (a 74F1779) and part of PLD U3. The counter is driven by TCLK. When the counter reaches terminal count, the filter clock output FILTCLK is toggled and the counter is reloaded with the byte in register U4. The counter load control signal, CLOAD, is generated asynchronously during the low half of TCLK when the counter terminal count signal TC is active. The design is such that for a load value of zero, FILTCLK will toggle every TCLK cycle, for a maximum output frequency of 4 MHz corresponding to a filter cutoff of 40 KHz. The toggle period of FILTCLK is n+1 times the TCLK period, where n is the value programmed into register U4, range {0..255}.

The signal CNTNABL is asserted active low to enable the filter clock divider, U5, and the control of FILTCLK by the counter terminal count output. CNTNABLis set active low by writing to ASIC bus address 1D (hex) and cleared by writing to ASIC bus address 1C (hex). Using this technique to control CNTNABL allowed this signal to be set and cleared under program control with no additional inputs to PLD U3.

For values of lowpass cutoff below about 160 Hz, the required period exceeds the capacity of the counter. So, for this case, one of the RTX2001 internal timers is used to generate periodic interrupts. The required period is loaded into timer 1 and timer 1 interrupts are unmasked. The timer 1 interrupt service routine consists of a write to ASIC bus address 1F (hex) which is decoded by PLD U3 to toggle the FILTCLK output if CNTNABL is high. This is probably the most obvious hardware/ processor resource tradeoff, where a hardware counter and latch was added to reduce interrupt servicing to a more reasonable frequency of about once every 32 milliseconds, particularly since the filter clock interrupt is at a lower priority than the DAC update interrupt.

External RAM

An additional 8K x 16 of static RAM has been added to the EBForth board for code and data space. The devices used are Cypress Semiconductor CY7C185 devices, U1 and U2, with a worst case access time of 45 ns. These RAMs have two chip select inputs, both of which must be active for accessing the chip. One is active high and one is active low. In order to minimize any delays that reduce read data access and write data setup times, the RTX2001 PCLK is connected directly to the active low chip select input, CE1. All other memory address decoding affects the other chip select, CE2.

The added RAM is memory mapped directly above the EBForth board ROM/RAM, starting at memory address 05000 (hex) through 08FFE. This RAM is byte addressable, with U1 containing the high byte and U2 the low byte. PLD U3 is used for decoding memory address bits MA[19:12], UDS, and LDS to generate the RAM select signals URAMSEL and LRAMSEL.

Miscellaneous Circuitry

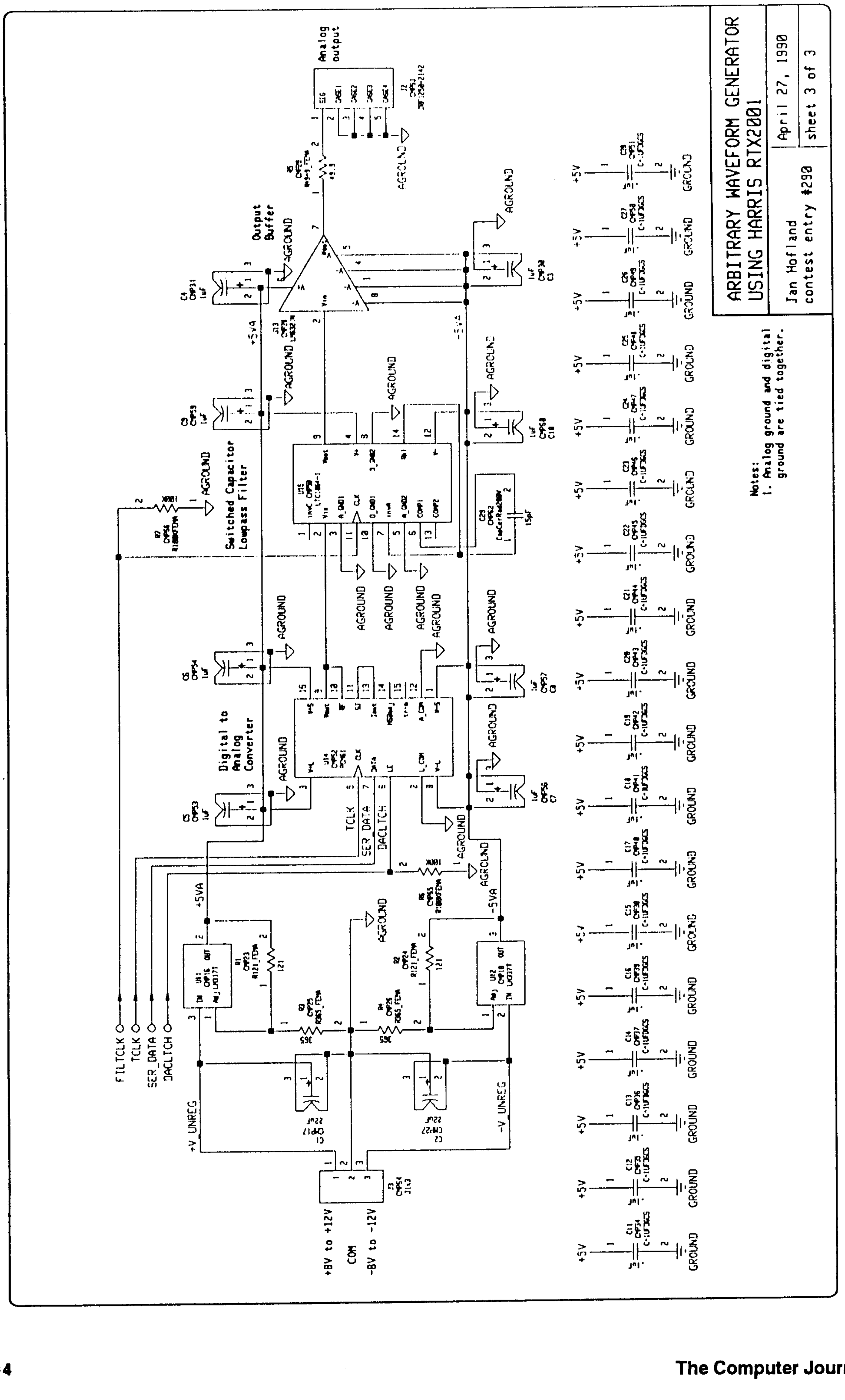
About the only hardware not already described is the RESET buffer, the NMI pushbutton interface, and the positive and negative voltage regulators for the analog circuitry. U16, a 74HC132 Schmitt trigger NAND gate, is used for the RESET interface and as an asynchronous set-reset flip-flop for the NMI pushbutton interface. The EBForth board RESET input is buffered by two sections of U16 and then input to the two PLDs, U3 and U8. One section of U8 is used to invert buffered RESET to drive the active low RESET inputs for the filter counter load register, U4, the input FIFO, U9 and U10, and the DAC shift register, U6 and U7.

Two sections of U16 form a cross-coupled set-reset flip-flop to debounce the current limited and can, therefore, tolerate being shorted to ground by the pushbutton switch. Among other things, it saves finding room for two pullup resistors. The output from U16C is connected directly to the non-maskable interrupt input on the EBForth board.

Adjustable three terminal regulators U11 and U12 are used to generate the +5V and -5V supply voltages for the DAC, the lowpass filter, and the output buffer.

Software Overview

The main software loop is a simple two command loop. It reads a command from the input FIFO and then executes that command. There are 16 recognized commands that are used


ARBITRARY WAVEFORM GENERATOR
USING HARRIS RTX2001
Jan Hofland
contest entry #290
April 27, 1990
sheet 3 of 3
Notes:
1. Analog ground and digital ground are tied together.
Analog output
Output Buffer
Switched Capacitor Lowpass Filter
Digital to Analog Converter
FILTCLK
TCLK
SER_DATA
DACLTCH
+8V to +12V
COM
-8V to -12V
+V UNREG
-V UNREG
+5VA
-5VA
AGROUND
GROUND
+5V

to set up the Waveform Generator for one of 5 operating modes, set the lowpass filter cutoff setpoint, set the desired DAC sample rate, and load data into the data queues, and set up the other parameters necessary. One of the commands is then used to start the output process.

There are supporting words that control how the next output point is calculated or fetched from the data queue and put into the DAC holding register ready for the next timer 0 interrupt. The output sample rate is controlled by programming it into timer 0. The timer 0 interrupt routine reads whatever data has been put into scratchpad register RH and writes it into the DAC shift register where it is automatically transferred into the serial DAC. Then the service routine decrements the RX register to set the stale data flag and returns to the routine just interrupted. That routine is a loop that repeatedly reads the RX register and when it detects a non-zero value, it updates the RH scratchpad register with the next data value to be output and clears the RX register.

Listing 2.

```
Here is the overall operating loop:

: main  ( -- )
     initialize
     BEGIN                     begin an infinite loop
     readFifo                  read a command from the input stream
     doCmd                     perform the command
     AGAIN ;                   go do it again

and here is the command execution word:

: doCmd ( n -- )              perform the command represented by n
       DUP cmdMsk AND          make sure that the upper 12 bits are zeroes
       IF                      if the result of the logical AND is non-zero
         DROP                  then it is an unrecognized command and discarded
         ." Unrecognized       Command "       an error message to the user
       ELSE parse EXECUTE      otherwise get an execution vector from the
```

There are software routines for creating and managing the queues in data address space and in code space. One of the more useful defining words implemented is one for building tables in code space. During compilation, the word table will create a table of n words, where n is the top stack value when table is invoked. During runtime, calling the table name will return the nth value in the table, where n is the top stack value. This little word is quite useful for building tables of execution vectors, for example. The remaining software is for controlling the specific devices attached to the ASIC bus and for calculating the interrelated parameters for sinewave and other output waveform generation.

In addition to the operating code, there is some software for demonstrating the current capabilities of the Arbitrary Waveform Generator.

See *Listing 1* for the specific commands which have been implemented and *listing* 2 for the overall operating loop and command execution word.

Software Description

ASIC Bus Addressable Devices

The following devices are addressable via the ASIC bus:

addr(hex)	read	write
18	(not assigned)	DAC Shift Register 16 bit
19	Input FIFO 16 bit	(not assigned)
1A	(not assigned)	(not assigned)
1B	FIFO Empty & Full Status bits 0 and 1	(not assigned)
1C	(not assigned)	Enable Filter Clock Counter
1D	(not assigned)	Disable Filter Clock Counter
1E	(not assigned)	Load Filter Clock Register 8 bits
1F	(not assigned)	Toggle Filter Clock Output

Registers 1C, 1D and 1F are pseudo registers in the sense that no data is stored. Instead, a particular action occurs by writing to that address. The Filter Clock Register, address 1E, uses only the lower 8 data bits. The upper 8 bits can be anything.

Sine Wave Generation

There are two methods of generating sinewave implemented. One of them is a direct table lookup, and one of them allows interpolation of values between the points represented in the table. The sine table is an 800 value table. The choice of 800 points isn't arbitrary. First, let's explain about how intermediate values of sine arguments are derived. The sine of the sum of two angles is expressed as follows:

```
sin(x + y) = sin(x) * cos(y) + cos(x) * sin(y)
```

Now, let the angle y be a small angle. Then we can approximate the sin(y) with y, where y is expressed in radians, and the cos(y) as nearly 1. So now the sine can be expressed as:

```
sin(x + y) = sin(x) + y * cos(x)
```

If we pick y to be a binary power of 2, then we can calculate the sin(x+y) with two table lookups, one shift to perform the multiply by y and an add. By choosing the interval between table points to be y = 1/128 radians, there are 804 points for one full cycle. This value was rounded to 800 to simplify some of the phase increment and sample rate calculations, without any loss in accuracy. The values in the table have been scaled to be equivalent to a 2 volt RMS output, allowing approximately 120 millivolts headroom to the DAC full scale output of 3 volts.

In the Fast sine mode of operation, the variable phase is used for phase accumulation and the variable phaseInc is used to hold the increment added to phase for each successive data point. The Forth word nextPt is used to look up the value of sine for the current phase, put that value in the RH scratchpad, and then update phase by adding the value of phaseInc to it.

In the accurate sine mode of operation, there are more software operations performed to determine the output value. There are two more variables used to keep track of the current value of phase and phase increment. They are delPhase and delPhaseInc. The Forth word newPhase takes care of calculating the new phase. It does so by first adding the incremental phase increment delPhaseInc to delPhase. This value takes care of the intermediate values between table points, and is scaled to one part in 800. Anything over 800 will contribute to the new value of phase, so the next value of phase = phase + phaseInc + delPhase/800. Now we can look up the sine of the phase variable from the table. Not done yet, though. Use the trigonometric identity that cos(x) = sin(x + pi/2) and look up the cosine.

Now all we must do is to multiply it by a scaled version of delPhase (our small angle) and add it to sin(x) to get the sine of the exact phase angle. The scaling needs to take care of converting to equivalent radians and divide by 128, as I discussed earlier. This calculation, and the appropriate scaling are performed in the Forth word cosAdj. The other two operations performed on the accurate sine point before it is put into the DAC scratchpad register, RH, is to multiply it by a scaling factor to set the output to some multiple of 50 mVRMS, and subtract an offset value. The word accuPt performs this operation. Finally, the word newAccuPt is the one that puts the value into scratchpad register RH, clears the stale data flag in RX, and proceeds to calculate the new value of phase and delPhase.

Interrupt Routines

There are three interrupts utilized in this design. The non-maskable interrupt is used to stop whatever is happening and abort the current operation with the message "Stopped". The word pbStop implements this routine. Timers 0 and 1 are used to control the DAC sample interval and the filter clock, respectively.

The timer 0 interrupt service routine is called newPt. It transfers the value in scratchpad register RH to the DAC shift register and then decrements the value in RX to make it non-zero.

The timer 1 interrupt service routine is even simpler. It just toggles the filter clock by writing to ASIC address 1F (hex). The timer 1 interrupt is used for filter cutoff values lower than about 160 Hz.

Parameter Setting Routines

The word setCutoff takes care of setting the filter clock value to some value between 1 Hz and 40 Khz. If the system clock divisor is less than 256, then the hardware divider is used and the timer 1 interrupt is disabled. On the other hand, if it is 256 or greater, then the hardware divider is disabled and the divisor is loaded into timer 1 and it is used for toggling the filter clock.

There is an inter-relationship between the system clock frequency, the desired sinewave output frequency, the number of points represented in one cycle, and the phase increment added to the phase variable for each output point. That relationship is:

```
phase_increment = (#table_points/cycle) * (timer 0 period)
                                   * frequency
                  ----------------------------------------
                  timer 0 input clock frequency (system clock)
```

It turns out that the ratio of #table_points/cycle to system clock is 1/10000. This is part of the reason for picking the number of points in the table as 800. The word calcInt finds the timer 0 period given the phase increment and the sinewave frequency. The word calcPhaseInc finds the required phase increment given the desired frequency. It uses a mode dependent value of minimum timer 0 period.

There is some interaction between the phase increment calculations and the timebase period, particularly for low frequency operation where the calculated value of phase increment is less than one. For this case, the timer 0 period is increased until a minimum value of phase increment can be achieved. The actual sinewave frequency is calculated from the phase increment and the timer 0 period. This value may differ slightly from the programmed frequency due to the discrete nature of the increment and the period. Such is the nature of approximating continuous values with digital representations.

Queue Building and Management

Queues are an important data structure used in this design. They are used for output data buffering and for simulation of the input FIFO. The queue building word for data space is buildDQ and for code space is buildCQ. Functionally, they behave the same. They're used by putting the desired size of the queue on the stack, followed by buildDQ or buildCQ, followed by the assigned queue name.

The queue structure consists of a variable to keep track of the number of values in the queue, the tail pointer where new values are written into the queue, the head pointer where data is read from the queue, an end pointer which points to the highest storage address in the queue, and 2n bytes of storage for the data. When the particular queues are defined, the head and tail pointers are initialized to point to the first storage location in the queue, the end pointer is set to point to the last storage location, and the #values variable is set to zero.

The data locations aren't initialized. They contain whatever random data was left in memory. The word >Cbuf is used for putting values into the queue at the tail pointer, incrementing the pointer and #values, and adjusting the pointer of it goes beyond the end pointer. The word Cbuf> extracts values from the queue at the head pointer, increments the pointer, decrements the #value variable, and adjusts the pointer if it exceeds the end pointer. The word >Que uses >Cbuf to put a value into the queue if the queue is not full. It returns a false flag if the queue wasn't full or a true flag if it was. The word Que> extracts a value from the queue if it isn't empty and returns a true flag if it was successful. It returns only a false flag if the queue was empty. The word #Que returns the number of valid values currently in the queue.

Arbitrary Data Output Modes

There are three output modes implemented using the data queues. The word circBuf simply extracts the next value from the current read queue and puts it into the RH scratchpad register. There is no checking to see if the queue is empty. This mode allows repeated output of the same data sequence and is useful for generating periodic waveforms. The word pingPong implements an output mode where data is read from the current read queue and output until the queue is emptied. Then the read buffer is switched to the other queue and it is read until it empties. When both queues have been emptied, it stops. The word onceOut operates in a similar manner, except that it stops after extracting data from the first queue.

We will look at the software source in the next issue.●

If we make peaceful revolution impossible, we make violent revolution inevitiable.—John F. Kennedy

B. Y. O. Assembler

Build Your Own (Cross-) Assembler...in Forth

by Brad Rodriguez

Introduction

In a previous issue of this journal I described how to "bootstrap" yourself into a new processor, with a simple debug monitor. But how do you write code for this new CPU, when you can't find or can't afford an assembler? Build your own!

Forth is an ideal language for this. I've written cross-assemblers in as little as two hours (for the TMS320, over a long lunch break). Two days is perhaps more common; and one processor (the Zilog Super8) took me five days. But when you have more time than money, this is a bargain.

In part 1 of this article I will describe the basic principles of Forth-style assemblers—structured, single-pass, postfix. Much of this will apply to any processor, and these concepts are in almost every Forth assembler.

In part 2, I will examine an assembler for a specific CPU: the Motorola 6809. This assembler is simple but not trivial, occupying 15 screens of source code. Among other things, it shows how to handle instructions with multiple modes (in this case, addressing modes). By studying this example, you can figure out how to handle the peculiarities of your own CPU.

Why Use Forth?

I believe that Forth is the easiest language in which to write assemblers.

First and foremost, Forth has a "text interpreter" designed to look up text strings and perform some related action. Turning text strings into bytes is exactly what is needed to compile assembler mnemonics! Operands and addressing modes can also be handled as Forth "words."

Forth also includes "defining words," which create large sets of words with a common action. This feature is very useful when defining assembler mnemonics.

Since every Forth word is always available, Forth's arithmetic and logical functions can be used within the assembler environment to perform address and operand arithmetic.

Finally, since the assembler is entirely implemented in Forth words, Forth's "colon definitions" provide a rudimentary macro facility, with no extra effort.

The Simplest Case: Assembling a NOP

To understand how Forth translates mnemonics to machine code, consider the simplest case: the NOP instruction (12 hex on the 6809).

A conventional assembler, on encountering a NOP in the opcode field, must append a 12H byte to the output file and advance the location counter by 1. Operands and comments are ignored. (I will ignore labels for the time being.)

In Forth, the memory-resident dictionary is usually the output "file." So, make NOP a Forth word, and give it an action, namely, "append 12H to the dictionary and advance the dictionary pointer."

```
HEX
: NOP,   12 C, ;
```

Assembler opcodes are often given Forth names which include a trailing comma, as shown above. This is because many Forth words—such as AND XOR and OR—conflict with assembler mnemonics. The simplest solution is to change the assembler mnemonics slightly, usually with a trailing comma. (This comma is a Forth convention, indicating that something is appended to the dictionary.)

The Class of "Inherent" Opcodes

Most processors have many instructions, like NOP, which require no operands. All of these could be defined as Forth colon definitions, but this duplicates code, and wastes a lot of space. It's much more efficient to use Forth's "defining word" mechanism to give all of these words a common action. In object-oriented parlance, this builds "instances" of a single "class."

This is done with Forth's CREATE and DOES>. (In fig-Forth, as used in the 6809 assembler, the words are <BUILDS and DOES>.) See *figure 1.*

In this case, the parameter (which is specific to each instance) is simply the opcode to be assembled for each instruction.

This technique provides a substantial memory savings, with almost no speed penalty. But the real advantage

Brad Rodriguez lives a double life. On odd-numbered days he is T-Recursive Technology, consulting in hardware and software design for real-time and embedded microprocessor applications. On even-numbered days he is a student, pursuing a Ph.D. in Electrical Engineering and exploring the possibilities of artificially-intelligent control systems. Brad discovered Forth in 1978, has been using it professionally since 1982, and has been known to annoy people with his incessant tales of how quickly things can be accomplished in Forth. He has written Forth assemblers for the 6809, 6801, 6502, Z8, Super8, TMS320,and two generally-unknown microprogrammed machines. The 6809 is his favorite 8-bit processor, partly because it and the 16-bit PDP-11 share the distinction of being "the best Forth processors that were not designed to be Forth processors." Brad prefers to be contacted as B.RODRIGUEZ2 on GEnie, but will accept email as bradford@maccs.dcss.mcmaster.ca on the Internet.

becomes evident when complex instruction actions—such as required for parameters, or addressing modes—are involved.

Handling Operands

Most assembler opcodes, it is true, require one or more operands. As part of the action for these instructions, Forth routines could be written to parse text from the input stream, and interpret this text as operand fields. But why? The Forth environment already provides a parse-and-interpret mechanism!

So, Forth will be used to parse operands. Numbers are parsed normally (in any base!), and equates can be Forth CONSTANTs. But, since the operands determine how the opcode is handled, they will be processed first. The results of operand parsing will be left on Forth's stack, to be picked up by the opcode word. This leads to Forth's unique postfix format for assemblers: operands, followed by opcode.

Take, for example, the 6809's ORCC instruction, which takes a single numeric parameter:

```
HEX
: ORCC,   1A C,  C, ;
```

The exact sequence of actions for ORCC, is: 1) put 1A hex on the parameter stack; 2) append the top stack item (the 1A) to the dictionary, and drop it from the stack; 3) append the new top stack item (the operand) to the dictionary, and drop it from the stack. It is assumed that a numeric value was already on the stack, for the second C, to use. This numeric value is the result of the operand parsing, which, in this case, is simply the parsing of a single integer value:

```
HEX
0F ORCC,
```

The advantage here is that all of Forth's power to operate on stack values, via both built-in operators and newly-defined functions, can be employed to create and modify operands. For example:

```
HEX
01 CONSTANT CY-FLAG          ( a "named" numeric value)
02 CONSTANT OV-FLAG
04 CONSTANT Z-FLAG
   ...
CY-FLAG Z-FLAG +  ORCC,      ( add 1 and 4 to get operand)
```

The extension of operand-passing to the defining words technique is straightforward.

Handling Addressing Modes

Rarely can an operand, or an opcode, be used unmodified. Most of the instructions in a modern processor can take multiple forms, depending on the programmer's choice of addressing mode.

Forth assemblers have attacked this problem in a number of ways, depending on the requirements of the specific processor. All of these techniques remain true to the Forth methodology: the addressing mode operators are implemented as Forth words. When these words are executed, they alter the assembly of the current instruction.

1. *Leaving additional parameters on the stack.* This is most useful when an addressing mode must always be specified. The addressing-mode word leaves some constant value on the stack, to be picked up by the opcode word. Sometimes

Figure 1

```
: INHERENT        ( Defines the name of the class)
    CREATE        ( this will create an instance)
       C,         ( store the parameter for each instance)
    DOES>         ( this is the class' common action)
       C@         ( get each instance's parameter)
       C,         ( the assembly action, as above)
    ;             ( End of definition)

HEX
12  INHERENT NOP,  ( Defines an instance NOP, of class
                        INHERENT, with parameter 12H.)
3A  INHERENT ABX,  ( Another instance - the ABX instr)
3D  INHERENT MUL,  ( Another instance - the MUL instr)
```

this value can be a "magic number" which can be added to the opcode to modify it for the different mode. When this is not feasible, the addressing-mode value can activate a CASE statement within the opcode, to select one of several actions. In this latter case, instructions of different lengths, possibly with different operands, can be assembled depending on the addressing mode.

2. *Setting flags or values in fixed variables.* This is most useful when the addressing mode is optional. Without knowing whether an addressing mode was specified, you don't know if the value on the stack is a "magic number" or just an operand value. The solution: have the addressing mode put its magic number in a predefined variable (often called MODE). This variable is initialized to a default value, and reset to this default value after each instruction is assembled. Thus, this variable can be tested to see if an addressing mode was specified (overriding the default).

3. *Modifying parameter values already on the stack.* It is occasionally possible to implement addressing mode words that work by modifying an operand value. This is rarely seen.

All three of these techniques are used, to some extent, within the 6809 assembler.

For most processors, register names can simply be Forth CONSTANTs, which leave a value on the stack. For some processors it is useful to have register names specify "register addressing mode" as well. This is easily done by defining register names with a new defining word, whose run-time action sets the addressing mode (either on the stack or in a MODE variable).

Some processors allow multiple addressing modes in a single instruction. If the number of addressing modes is fixed by the instruction, they can be left on the stack. If the number of addressing modes is variable, and it is desired to know how many have been specified, multiple MODE variables can be used for the first, second, etc. (In one case—the Super8—I had to keep track of not only how many addressing modes were specified, but also where among the operands they were specified. I did this by saving the stack position along with each addressing mode.)

Consider the 6809 ADD instruction. To simplify things, ignore the Indexed addressing modes for now, and just consider the remaining three addressing modes: Immediate, Direct, and Extended. These will be specified as follows:

	source code	assembles as
Immediate:	number # ADD,	8B nn
Direct:	address <> ADD,	9B aa
Extended:	address ADD,	BB aa aa

Since Extended has no addressing mode operator, the mode-variable approach seems to be indicated. The Forth words # and <> will set MODE.

Observe the regularity in the 6809 opcodes. If the Immediate opcode is the "base" value, then the Direct opcode is this value plus 10 hex, and the Extended opcode is this value plus 30 hex. (And the Indexed opcode, incidentally, is this value plus 20 hex.) This applies uniformly across almost all 6809 instructions which use these addressing modes. (The exceptions are those opcodes whose Direct opcodes are of the form 0x hex.)

Regularities like this are made to be exploited! This is a general rule for writing assemblers: find or make an opcode chart, and look for regularities—especially those applying to addressing modes or other instruction modifiers (like condition codes).

In this case, appropriate MODE values are suggested:

```
VARIABLE MODE  HEX
: #        0 MODE ! ;
: <>      10 MODE ! ;
: RESET   30 MODE ! ;
```

The default MODE value is 30 hex (for Extended mode), so a Forth word RESET is added to restore this value. RESET will be used after every instruction is assembled.

The ADD, routine can now be written. Let's go ahead and write it using a defining word:

```
HEX
: GENERAL-OP   \ base-opcode -
     CREATE C,
     DOES>      \ operand -
     C@              \ get the base opcode
      MODE @ +        \ add the "magic number"
     C,              \ assemble the opcode
     MODE @ CASE
        0 OF C, ENDOF      \ byte operand
       10 OF C, ENDOF      \ byte operand
       30 OF , ENDOF       \ word operand
     ENDCASE
     RESET ;
8B GENERAL-OP ADD,
```

Each "instance" of GENERAL-OP will have a different base opcode. When ADD, executes, it will fetch this base opcode, add the MODE value to it, and assemble that byte. Then it will take the operand which was passed on the stack, and assemble it either as a byte or word operand, depending on the selected mode. Finally, it will reset MODE.

Note that all of the code is now defined to create instructions in the same family as ADD:

```
HEX 89 GENERAL-OP ADC,
    84 GENERAL-OP AND,
    85 GENERAL-OP BIT,
          etc.
```

The memory savings from defining words really become evident now. Each new opcode word executes the lengthy bit of DOES> code given above; but each word is only a one-byte Forth definition (plus header and code field, of course).

This is not the actual code from the 6809 assembler—there are additional special cases which need to be handled. But it demonstrates that, by storing enough mode information, and by making liberal use of CASE statements, the most ludicrous instruction sets can be assembled.

Handling Control Structures

The virtues of structured programming, have long been sung—and there are countless "structured assembly" macro packages for conventional assemblers. But Forth assemblers favor label-free, structured assembly code for a pragmatic reason: in Forth, it's simpler to create assembler structures than labels!

The structures commonly included in Forth assemblers are intended to resemble the programming structures of high-level Forth. (Again, the assembler structures are usually distinguished by a trailing comma.)

1. BEGIN, ... UNTIL,

The BEGIN, ... UNTIL, construct is the simplest assembler structure to understand. The assembler code is to loop back to the BEGIN point, until some condition is satisfied. The Forth assembler syntax is

```
BEGIN,    more code     cc UNTIL,
```

where 'cc' is a condition code, which has presumably been defined—either as an operand or an addressing mode—for the jump instructions.

Obviously, the UNTIL, will assemble a conditional jump. The sense of the jump must be "inverted" so that if 'cc' is satisfied, the jump does NOT take place, but instead the code "falls through" the jump. The conventional assembler equivalent would be:

```
xxx:  ...
      ...
      ...
      JR  -cc,xxx
```

(where ~cc is the logical inverse of cc.)

Forth offers two aids to implementing BEGIN, and UNTIL,. The word HERE will return the current location counter value. And values may be kept deep in the stack, with no effect on Forth processing, then "elevated" when required.

So: BEGIN, will "remember" a location counter, by placing its value on the stack. UNTIL, will assemble a conditional jump to the "remembered" location.

```
: BEGIN, ( - a)        HERE ;
: UNTIL, ( a cc - )   NOTCC  JR, ;
```

This introduces the common Forth stack notation, to indicate that BEGIN, leaves one value (an address) on the stack. UNTIL, consumes two values (an address and a condition code) from the stack, with the condition code on top. It is presumed that a word NOTCC has been defined, which will convert a condition code to its logical inverse. It is also presumed that the opcode word JR, has been defined, which will expect an address and a condition code as operands. (JR, is a more general example than the branch instructions used in the 6809 assembler.)

The use of the stack for storage of the loop address allows BEGIN, ... UNTIL, constructs to be nested, as:

```
BEGIN, ... BEGIN, ... cc UNTIL, ... cc UNTIL,
```

The "inner" UNTIL, resolves the "inner" BEGIN, forming a loop wholly contained within the outer BEGIN, ... UNTIL, loop.

2. BEGIN, ... AGAIN,

Forth commonly provides an "infinite loop" construct, BEGIN ... AGAIN , which never exits. For the sake of completeness, this is usually implemented in the assembler as well.

Obviously, this is implemented in the same manner as BEGIN, ... UNTIL, except that the jump which is assembled by AGAIN, is an unconditional jump.

3. DO, ... LOOP,

Many processors offer some kind of looping instruction. Since the 6809 does not, let's consider the Zilog Super8; its Decrement-and-Jump-Non-Zero (DJNZ) instruction can use any of 16 registers as the loop counter. This can be written in structured assembler:

```
DO,    more code   r LOOP,
```

where r is the register used as the loop counter. Once again, the intent is to make the assembler construct resemble the high-level Forth construct.

```
: DO,   ( - a)       HERE ;
: LOOP, ( a r - )    DJNZ, ;
```

Some Forth assemblers go so far as to make DO, assemble a load-immediate instruction for the loop counter—but this loses flexibility. Sometimes the loop count isn't a constant. So I prefer the above definition of DO, .

4. IF, ... THEN,

The IF, ... THEN, construct is the simplest forward-referencing construct. If a condition is satisfied, the code within the IF,...THEN, is to be executed; otherwise, control is transferred to the first instruction after THEN,.

(Note that Forth normally employs THEN, where other languages use "endif." You can have both in your assembler.)

The Forth syntax is

```
cc IF, ... ... ... THEN,
```

for which the "conventional" equivalent is

```
     JP   ~cc,xxx
     ...
     ...
     ...
xxx:
```

Note that, once again, the condition code must be inverted to produce the expected logical sense for IF, .

In a single pass assembler, the requisite forward jump cannot be directly assembled, since the destination address of the jump is not known when IF, is encountered. This problem is solved by causing IF, to assemble a "dummy" jump, and stack the address of the jump's operand field. Later, the word THEN, (which will provide the destination address) can remove this stacked address and "patch" the jump instruction accordingly.

```
: IF, ( cc - a)   NOT 0 SWAP JP,  ( conditional jump
                  HERE 2 - ;        with 2-byte operand)
: THEN, ( a)   HERE SWAP ! ;      ( store HERE at the
                                  stacked address)
```

IF, inverts the condition code, assembles a conditional jump to address zero, and then puts on the stack the address of the jump address field. (After JP, is assembled, the location counter HERE points past the jump instruction, so we need to subtract two to get the location of the address field.) THEN, will patch the current location into the operand field of that jump.

If relative jumps are used, additional code must be added to THEN, to calculate the relative offset.

5. IF, ... ELSE, ... THEN,

A refinement of the IF,...THEN, construct allows code to be executed if the condition is NOT satisfied. The Forth syntax is

```
cc IF, ... ... ELSE, ... ... THEN,
```

ELSE, has the expected meaning: if the first part of this statement is not executed, then the second part is.

The assembler code necessary to create this construct is:

```
     JP   ~cc,xxx
     ...            ( the "if" code)
     ...
     JP   yyy
xxx: ...            ( the "else" code)
     ...
yyy:
```

ELSE, must modify the actions of IF, and THEN, as follows: a) the forward jump from IF, must be patched to the start of the "else" code ("xxx"); and b) the address supplied by THEN, must be patched into the unconditional jump instruction at the end of the "if" code ("JP yyy"). ELSE, must also assemble the unconditional jump. This is done thus:

```
: ELSE ( a - a)   0 T JP,     ( unconditional jump)
                  HERE 2 -    ( stack its address
                                for THEN, to patch)
                  SWAP        ( get the patch address
                                of the IF, jump)
                  HERE SWAP ! ( patch it to the current
                                location, i.e., the
                                   next instruction)
                  ;
```

Note that the jump condition 'T' assembles a "jump always" instruction. The code from IF, and THEN, can be "reused" if the condition 'F' is defined as the condition-code inverse of 'T':

```
: ELSE ( a - a)   F IF,  SWAP  THEN, ;
```

The SWAP of the stacked addresses reverses the patch order, so that the THEN, inside ELSE, patches the original IF; and the final THEN, patches the IF, inside ELSE,. Graphically, this becomes:

```
IF,(1) ... IF,(2)  THEN,(1) ... THEN,(2)
           \_______________/
             inside ELSE,
```

IF,...THEN, and IF,...ELSE,...THEN, structures can be nested. This freedom of nesting also extends to mixtures of these and BEGIN,...UNTIL, structures.

6. BEGIN, ... WHILE, ... REPEAT,

The final, and most complex, assembler control structure

See BYO Assembler, page 39

Assembly Language Programming

Today's Example: ZCNFG

by A. E. Hawley

The last two articles were about 'getting started' in assembly language programming. Now lets take a look at the real world of AL programs with a tour of ZCNFG, a public domain program that is sophisticated enough to illustrate many of the principles that you can apply to your own programs. The main focus will be on a discipline that most of us learned the hard way by ignoring it: source program structure. Along the way, we'll get introduced to some of the algorithms and code structures in ZCNFG. They will help to dispel some of the mystery about writing .CFG files for other programs.

To get the most out of the following discussion, you should have the complete source for ZCNFG available for study and reference. ZCNFG is available via modem download from Z-nodes and via mail from *ZSUS*.

What ZCNFG Does

Many commercial programs come with an INSTALL utility. That utility customizes the program so that it is compatible with your computer hardware and your performance preferences. For example, a communications program needs to know the port assigned to the modem, and the default communication protocol to use. CP/M programs that manipulate the screen need installation of the control data for your terminal or video display. INSTALL utilities configure their target program by modifying data in the image of the program stored on disk so that subsequent invocations will exhibit the configured performance. Each installation program is written by a different author, and no two user interfaces are alike. Configuration of public domain programs has been traditionally handled by reassembly of the program or by changing the affected data with a debugger or a file patching program like ZPATCH. Both require obvious skills on the part of the user as well as a certain amount of research to discover what changes are possible and what new values are required to achieve the desired changes. Again, no two programs are configured the same way!

During early development of ZMAC, ZML, and ZMLIB I wrote configuration utilities for each program. Each enhancement could result in a revision of the associated configuration program, including the inevitable debugging sessions. What a lot of wheel-spinning! The first solution was to combine the three programs into one, with a common main program to maintain an on-screen menu and three data sections to specify what is displayed on the menu. Of course, there was also a section to read and write the proper file being configured. Since this made a rather large configuration program, the next move was to make three files of the tables. The configuration program then loaded only the file appropriate for the target program. Now all I had to do was maintain the file of configuration data for each program.

There are practical pay-offs to well thought out organization of the code in a program!

There is only one configuration program to maintain, and debugging caused by changes in the target file is eliminated. Now *that* is worthwhile!

ZCNFG was born when I realized that this idea could be used with *any* target program that is organized with configuration data near the start of the program. Although many older CP/M programs were not written that way, more recent programs adapted or written for Z-system have adopted that style. There are practical pay-offs to well thought out organization of the code in a program!

ZCNFG provides a menu driven means of setting default options in executable programs, including ZCNFG itself. It does this with the help of a configuration file unique to each target program. For example, the configuration file for ZCNFG is ZCNFG.CFG and that for FF24.COM is FF24.CFG.

The code in ZCNFG parses the command line to determine what file is to be configured, and which .CFG file to use. It performs the initialization functions required, and provides for an EXIT routine which is the *one* point at which the program returns control to the operating system. A few lines showing ZCNFG commands at the bottom of the menu display are maintained by code in ZCNFG, rather than by data in the .CFG file.

The .CFG file contains four kinds of data for each menu:

1. *Menu Data Structure:* a list of 5 pointers
2. *Case Table:* a data record for each configuration item
3. *Screen Image:* a prototype menu screen image
4. *Help Screen:* a block of text explaining menu items

A. E. (Al) Hawley started out as a Physical Chemist with a side line love of electronics when it was still analog. He helped develop printed circuit technology, and contributed to several early space and satellite projects. His computer experience started with a Dartmouth Time-Share system in BASIC, FORTRAN, and ALGOL. His first assembly language program was the REVAS disassembler, written for a home-brew clone of the Altair computer. As a member of the ZCPR3 team, he helped develop ZCPR33 and became sysop of Z-Node #2. He has contributed to many of the ZCPR utilities, and written several. He is author of the ZMAC assembler, ZML linker, and the popular ZCNFG utility.

These data structures are described in detail in ZCNFG.WS, included in ZCNFG17.LBR. You did get it, didn't you?

ZCNFG reads the current values of configurable items from the target file, uses data from the CFG file to display them in the menu, allows the user to change them and then updates the data in the target file. After exiting ZCNFG, the program will now behave in accord with the newly chosen options because the data in its configuration area has been rewritten. This process is much easier and faster than making the same changes with a debugger or file patcher.

Figure 1.
A Typical Configuration Menu

```
                         FF CONFIGURATION MENU

     Drives which may be searched by FF. Type drive letter to toggle.

                      A B C D E F G H I J K L M N O P
                      Y Y Y Y

            R) Return number of files found in register number  10
            S) include System files in the search ?             NO
            T) Terminator following DIR/DU for found files is    >
            V) View console screen with Paging enabled?        YES
            W) auto-Wildcarding in the search argument?        YES
            0) Limit default drives to logged-in ones?          NO
            1) display drive # headers for found files?         NO
            2) Use NDR to control search?                      YES
            3) Wheel control? YES    4) NDR for wheel user?     NO

                       ZCNFG INSTALLATION CONTROL
   X or Esc =Save changes & eXit      Q,^C =Quit with no changes saved
 / or ? =Explain Options      > or . =Next Menu      < or , =Previous Menu
       Which choice?
```

Figure 1 shows a typical configuration menu screen. The user types the character associated with one of the items on the menu. If that item has a YES/NO choice, then the current value is toggled and redisplayed. If, like item R), a numeric value is required a prompt will appear asking for input of that value. The prompt may indicate the allowable range of values; only values within that range will be accepted. An explanation of the function of each configuration choice may be seen by pressing the ? or / key.

Listing 1.

```
                PROGRAM CODE

ZCNFG:  JP      BEGIN

Z3MARK: DB      'Z3ENV'         ;identifies program as ZCPR3x utility
        DB      1               ;external environment
Z3ENV:  DW      0               ;this address set by Z3INS or ZCPR33/4
        DW      ZCNFG           ;compatible with type 4 enviroment

;configuration block for THIS program.
CNFGID: DB      'ZCNFG'         ;ID string, null terminated
        DS      4,0             ;max 8 char plus null terminator
ALTUSR: DB      -1              ;-1 = search default user
;additional configuration items go here
```

Functional Organization

An AL program is usually organized as shown in *figure* 2. This arrangement, though common, is certainly not mandatory. Experience has shown that the program header should occur first in the source file, and that it is convenient to put most of the data items at the end of the file. A program like ZCNFG which is expected to be executable after assembly/link must observe certain conventions at the start of the code in order to be compatible with the operating system. Likewise, the position of the block of configuration data is predicated on the use of ZCNFG or on convenience when using a debugger to set default values.

At the very start of a program is the *Program Header*. This header is in two basic parts: textual information that identifies the program to the programmer, and definition of quantities to be used by the assembler and linker.

The textual information takes the form of comments in which each line starts with a semicolon. It is ignored by the assembler; if you leave it out you are only cheating yourself (or others) of some information.

Quantities used by the assembler and linker defined here are Constants, Public Symbols, and External Symbols. The assembler does not care about the order in which these symbols are declared, as long as the statements themselves are correct. You will note, however, in the ZCNFG source code, the .REQUEST statements which name libraries like Z3LIB, SYSLIB, and others. The order in which these statements occurs *is important!* The linker searches those libraries in the order in which they are named. These .request statements are a substitute for including the library names in the linker command tail. ZMAC, M80, and the SLR assemblers recognize the .request statement; others may not, in which case such statements must be removed and the libraries included in the linkers list. If you get the order wrong the linker will complain to you about 'undefined symbols' or unresolved externals.

Constants used by the assembler are defined by EQU statements. Such statements might, for example, define ASCII constants like CR and LF. This is a convenient place to define version number and version date. Other constants defined this way in ZCNFG are standard system addresses like the targets of the WB and BDOS jumps in page 0 and the system buffers (FCB and TBUF) in page 0. For ZCNFG, the offsets to certain important data in the target program are defined this way.

Symbols which are to be shared with other program modules are declared PUBLIC; those which the current program

```
Listing 2.
                STARTUP

BEGIN:  LD      (STACK),SP      ;save system stack pointer
        LD      SP,STACK        ;set up local stack
```

needs from some other module are declared EXTernal. The assembler places PUBLIC symbols and their address value in the .REL file. It also places EXT symbols in the REL file along with information about where in the current module each symbol is used. The linker, after it knows the values for all the PUBLIC symbols for *all* the modules being linked, puts these values in the appropriate EXT locations. For each PUBLIC symbol in the current module there may be many EXT declarations in *other* modules. Similarly, for each EXT symbol in the current module there must be a matching PUBLIC symbol in one of the other modules.

The program header is also a good place to define macros that will be used in the program. ZCNFG defines three such macros in this section. Macros do not produce code at this point; actual code is only produced when the macro is named later on in the program as an instruction.

When programs get complex, the definitions section can get large enough to be an inconvenience in editing and reading the source file. You can make things appear a little simpler by transferring as much of this section as you like to a separate file, then including it in the assembly source with a MACLIB or INCLUDE statement. For example, if all the header except the opening comment lines of ZCNFG had been removed and placed in a file called ZCNFGHDR.LIB, then the following statement would take its place in the source file:

```
    INCLUDE     ZCNFGHDR
```

Assembly language code, as you will have observed already, includes a great deal of excruciating detail which for most people interferes with comprehension of the main program flow and functions. The INCLUDE provides a way of 'hiding' such detail. Many HLLs also provide the same facility, and for the same reason! The assembler doesn't care which way you choose; the code produced is the same either way.

Program Code

Most program code starts with a JP instruction whose target is the actual start of program related code. This strategy permits convenient inclusion of a data area at the beginning of the program. If a program is intended to be a Z-system utility, it must start with the standard ZCPR header. If it is intended to be configured using all the facility of ZCNFG, there must be an immediately following block of header and

Figure 2.

```
                    PROGRAM STRUCTURE

PROGRAM HEADER
   Program comments - Name, Author, Date, Purpose, References
   Program definitions (<variable> EQU <value>)
   External declarations (in Z3LIB, SYSLIB, etc.)
   Public declarations

PROGRAM CODE (see listing 1.)
   JP START (or RST 0, DW START or RET, DW START)
   Z34 ENV Structure
   Configuration Block

STARTUP (see listing 2.)
   Check for Z80 cpu, exit with message if not.
   Save SP and set local stack.

ENVIRONMENT INITIALIZATION (see listing 3.)
    Identify OS parameters of importance to the program
       OS version - CP/M 2 or 3? ZRDOS? ZSDOS? Z3PLUS? ..etc.
       CCP identification - CP/M or ZCPR? Which version of ZCPR?
       Is DateStamping Available? Which kind? (DS, ZSDOS, CPM+)
       Is a Real Time Clock available (other than DS, ZSDOS, CPM+)?
    Initialize the ZCPR and VLIB Environment if appropriate.
    Error Exit if OS support is inadequate

HELP (see listing 3.)
   Parse CL for help request.
   Display HELP screen if required by a CALL to a Print_Help routine
   Return to OS by a jump to PROGRAM EXIT if help was printed,
   Else continue with ...

PROGRAM INITIALIZATION (see listing 3.)
    Copy configuration data from the CFG block to a data buffer or
     final destination.
    Parse command line for options and update those in cfg data buffer.
    Allocate memory usage
    Initialize buffers and data values
    Initialize FCBs
    Open input file(s) if appropriate
    Open output file(s) if appropriate

MAIN PROGRAM (see listing 4.)
   Here is the place for your program to "do it's thing".

PROGRAM EXIT
    close any open files
    relog current directory if appropriate
    Restore any changed OS parameters
    Restore callers stack pointer from saved value
    exit via RET if CCP has not been overwritten,
      else exit via JP 0 or RST 0.

SUBROUTINES
    Print_Help is a called routine so that it can be invoked
      from anywhere in the program without aborting to the OS.
    Initialization routines are conveniently placed here
      and called from the initialization section to avoid
      cluttering up the main flow of the program.
    Special file handling routines
    Screen management routines
    Low level functions like data format conversion routines
    BDOS function routines

DATA (Initialized)
    Messages
    Screen Images
    File Control Blocks
    Initialized variables

DATA (Uninitialized)
    Stack space
    Callers stack pointer
    Copy of configuration data
    Uninitialized variables
    Buffers
```

data for use by ZCNFG. CFG files do not start with a JP instruction because when assembled they are data files and are not executable. In fact, they start with a RST 0 instruction which simply executes a warm boot and return to the CCP. This protects the file from inadvertent execution. See *Listing 1* for an example.

Listing 3.

```
                HELP & INITIALIZATION

        CALL    INIT            ;set current du, Test for Z3
        LD      DE,SIGNON
        CALL    TYPLIN
        CALL    HELP            ;provide help if requested & quit
        CALL    PGMINI          ;Get file spec from FCB, open file
                                ;abort with message if bad file spec
        CALL    FILINIT         ;identify & load the overlay file
        CALL    SCR_LD          ;load screen image(s)
```

Startup

This is, apart from the initial jump, the beginning of the executable code in the program. In ZCNFG, the label at this point is BEGIN:, and the initial jump instruction is JP BEGIN. If you are concerned about the possibility of your program being executed on a CPU that doesn't support the code in the program (i.e. Z80 code and an 8080 CPU), then here is the place for code that identifies the CPU. If the wrong CPU is present, then the code must immediately exit. An error message is appropriate. The code up to this point must use only instructions supported by both CPUs (8080 code in this case). Remember, we are talking about the *instructions* in machine language generated, *not* the assembler mnemonics used! For example, a JP instruction will execute properly on both CPUs but a JR instruction will only be recognized by the Z80. Several public domain programs use this kind of test; ZCNFG does not.

We are now (almost) ready to start computing. First, we have to make sure the Stack Pointer is being properly managed. This one item is probably responsible for more system crashes than all others combined! When your program starts executing, the stack currently assigned is that of the program (usually the CCP) that called it. That stack may or may not have enough room to support the requirements of your program. If it is too small, your program is guaranteed to crash the system. If you prefer not to play russian roulette, do like ZCNFG does at the first opportunity; save the SP. Define a local stack within the program. *Listing 2* shows an example.

Help and Initialization

Two initialization sections occur in ZCNFG. The first is the CALL INIT instruction. INIT installs the current version number and date in the signon message, then sets a flag if ZCPR3/33/34 is available. This portion of the initialization provides information for the following signon message display and the call to the program HELP function. When the HELP routine determines that a help screen is not appropriate it simply returns; otherwise, the help screen is printed and the program is terminated by a jump to the main exit routine (its name in ZCNFG is QUIT:) which restores the SP and returns control to the CCP.

The second initialization section comprises the calls to PGMINI, FILINIT, and SCR_LD. PGMINIT performs a number of housekeeping tasks, as outlined in *figure* 2. Here, the target filespec is determined and the file is opened for reading. The default name for the CFG file is defined if it isn't present on the command line.

FILINIT reads the first page of the target program. If it's a ZCPR34 type 4 program, then a second read is performed to get the real first page (256 bytes) of the program where the configuration data resides. This data is loaded into a 256 byte buffer named TGTBUF. It then looks at offset 10Dh in the configuration page for the name of a CFG file. FILINIT selects the first CFG filespec from the following prioritized list:

1. the second argument in the command tail
2. the name (if any) from the target configuration page
3. the default defined in PGMINIT

The CFG file is loaded into memory starting at the first unallocated location above ZCNFG. The address of the start of this free memory is stored in a variable labeled OVRLAY:.

Listing 4.

```
                MAIN PROGRAM

;select & set options interactively
;This is a loop whose exit is one of the cases
SETOPT: CALL    Z3CLS           ;clear screen
        LD      DE,(SIMAGE)     ;-> screen image
        CALL    TYPLIN          ;display the screen
        LD      A,(Z3MSGF)
        OR      A               ;ZCPR3 present?
        JR      Z,NOT1Z3
        CALL    AT              ;cursor positioning if Z3
ATPRPT: DB      19,1            ;prompt near screen bottom
NOT1Z3: LD      DE,PRMPT0       ;-> prompt line(s)
        CALL    TYPLIN          ;display user prompt

;get user input. Make changes as requested, update the
;screen image and the target configuration block.
GETINP: CALL    CIN             ;wait for & get user input
        JR      Z,GETINP        ;z = no input yet
        CALL    UCASE           ;make upper case
        CALL    ISPRINT         ;printable character?
        CALL    Z,COUT          ;echo it if so
        LD      HL,(CASTBL)     ;->case table
        CALL    MCASE           ;do case, ret to here if no exit
        JR      C,GETINP        ;on bad cmd, repeat prompt & retry
        JR      SETOPT          ;redisplay screen & prompt after update
```

This location is also labeled $MEMRY:. When ZCNFG was assembled and linked, the linker stored the location of free memory there. *The label '$MEMRY' is reserved for this use by most modern linkers.* The other label for this location, OVRLAY, refers to the same quantity and is used for semantic and debugging convenience.

See Assembler Programming, page 56

The NZCOM IOP

A Background Clock Display

By Terry Hazen

Introduction

I've always wanted to have a background clock display available on my terminal. While I was working on the RSX version of the HP14 integer RPN calculator, Joe Wright commented that it would really make more sense to do a calculator as an IOP. I hadn't really thought much about IOPs before. My Ampro BIOS didn't support IOPs and I hadn't seen much written about them. Since NZCOM makes it so easy to create an operating system that can include an IOP, I thought it would be fun to learn more about them by adapting HP14 as an IOP calculator. I did (see HP14.LBR, on your favorite Z-node), and I liked it so well I decided to try integrating a background clock display into the operating system as an IOP module.

In this first of two articles, I'll present CLKIOP, a background NZCOM IOP clock display module. Next time I'll present IOPLDR, a general-purpose NZCOM IOP module loader that is combined with an IOP REL module and some module-specific routines to create a stand-alone COM file that will load, control and remove the IOP module. We'll use it to create the stand-alone IOP clock display utility IOPCLK.COM.

Before we look at the IOP clock, though, let's take a quick look at the two most common ways of extending the CP/M operating system, the RSX and the IOP. Both RSX and IOP modules are used to extend the CP/M operating system in various ways. They can both intercept and modify BDOS or BIOS functions and even provide new BDOS or BIOS functions.

The RSX

The RSX (Resident System Extension) is a structure that is loaded under the CCP and protected from being overwritten by warm boots. The RSX can be easily loaded or removed to provide specific functions on a temporary or semi-permanent basis and can be made relatively large if required. Since an RSX is loaded below the protected CCP, however, the available TPA space is reduced by the size of the RSX plus the size of the CCP. The extra amount of space an RSX takes can make it relatively unattractive for more permanent uses in systems that are short on TPA.

Bridger Mitchell covers the details of the standard Plu*Perfect CP/M 2.2 RSX structure in his *Advanced CP/M* column in *TCJ* 34. Al Hawley describes ZREMOTE, a practical RSX application, in his *Getting Started in Assembly Language* column in *TCJ* 50.

Figure 1.

```
Tracing an NZCOM BIOS I/O Call

  +--> CBIOS Jump Table--> CBIOS I/O Routine
  |
+-|--> IOP Input Jump Table--> (Intercepts)--> IOP Output Jump Table--+
| |                                                                   |
| +----NZBIOS Auxillary IOP Jump Table <------------------------------+
+------NZBIOS Main Jump Table <-------------+
                                            |
       Applications Program BIOS I/O Call---+
```

The IOP

The IOP (Input/Output Package) is another way to extend the CP/M system. In contrast to the RSX, an IOP module is dynamically loaded into an IOP buffer in high memory and reduces the available TPA space only by the size of the IOP buffer itself. It works by intercepting and modifying BIOS I/O calls. Its size is limited to the currently defined size of the IOP buffer, although with NZCOM, that may be easily changed as required.

The standard IOP structure provides two jump tables to make it easy to intercept and modify of BIOS I/O calls (*Figure 1.*) Calls are passed from the main NZBIOS jump table to the IOP input jump table, which directs them to the IOP output jump table, which in turn directs them back to an auxiliary NZBIOS IOP jump table, which directs them to the CBIOS main jump table, where they are finally

Terry Hazen has a background in analog electronic and mechanical engineering. He is currently a product design consultant, specializing in medical electronic systems. He encountered his first computer in the 1960's and got very frustrated trying to write small punched-card batch-processed ALGOL programs. He got his first Z80 computer in 1982 and has been pursuing Z80 hardware and software projects ever since. His company, n/SYSTEMS, produces the MDISK 1 megabyte add-on RAM disk for Ampro LB computers. MDISK also provides the Ampro with bank-switching capabilities for operating system expansion. Terry enjoys designing and building varied types of hardware and software projects, not all of them computer-related. His recent software projects include the HP and HPC RPN calculators and the REMIND appointment reminder utility as well as upgrades to his SCAN text file viewing utility and ZP file/disk/memory record patcher. He may be reached by voice at (408)354-7188 or by message on Ladera Z-node #2. His address is 21460 Bear Creek Road, Los Gatos, CA 95030.

passed on to the actual working routines in the CBIOS. Whew! Intercepting a BIOS I/O call in the IOP module is easy. You only have to change the call's target address in the IOP input jump table to point to a local IOP interception routine.

The standard IOP structure also has several additional entry points to optional custom routines that can provide other I/O services in a standard way:

The STATUS routine at IOP+0 can be used to tell the calling program information about the current IOP module.

The SELECT routine at IOP+3 can be used to assign physical to logical devices or to deselect the current IOP module.

The NAMER routine at IOP+6 can return a pointer to the name of the IOP module.

The IOPINIT routine at IOP+9 is always called by the loader during the loading process and performs all required IOP initiation.

The NEWIO routine at IOP+21h can be designed to install new I/O drivers.

Rick Conn covers the original IOP structure and his intended uses for each of these IOP routines in *'ZCPR3 The Manual'* (New York Zoetrope, 1985). p212ff.

In NZCOM, Joe Wright extended the original IOP concept to include the IOP module as a preprocessor for BIOS I/O calls and designed a new standard IOP data structure to make the IOP easier to use. *He* also extended the standard IOP structure to support several more optional but useful I/O routines especially designed for I/O redirection:

COPEN at IOP+24h and CCLOSE at IOP+27h can be used to open and close a file specified as the CON: disk file.

LOPEN at IOP+2Ah and LCLOSE at IOP+2Dh can be used to open and close a file specified as the LST: disk file.

These routines can be designed to control files used by IOP I/O redirection packages such as Alpha Systems' NuKey, BPRINTer and RECORDer.

The IOP Clock

A good background clock display IOP routine should run constantly in the background, updating a date and time display somewhere on the terminal screen while keeping out of the way of foreground application utilities. But where should the clock display go? I first tried putting it in the upper right corner of the screen display area and immediately ran into problems because it took so long to constantly save and restore the current cursor position. It also turns out that GETCUR, the VLIB routine to report the current cursor position, only works with one type of terminal and isn't a general routine. For example, it doesn't work at all with ANSI terminals. Applications programs also bumped the clock display off the screen from time to time and it often ended up being overlaid on a part of the applications program display screen. All in all, it wasn't very satisfactory. Close, but no cigar.

Then I made the discovery that all my terminals had some form of host computer message field on the top line of the display, just above the regular display area. As it turned out, the message field proved to be an excellent place for a clock display. On the minus side, not all older terminals have message fields. Also, since each terminal

Listing 1

```
; Module:       CLKIOP
; Author:       Terry Hazen
; Date:         03/22/91
; Version:      1.0
;
; Note:  This IOP module conforms to Joe Wright's Standard
; Z-System I/O Package structure, version 4.0, May 4, 1989.
;
; Equates
;
bell    equ     07
on      equ     0ffh
;
; We need one library routine:
;
        .request        syslib
        ext             ma2hc

;=============================================================
;
; The following jumps and package ID are a fixed data
; structure and cannot be changed if this package is to
; work with standard loaders and SHOW programs:
;
; IOP Input Jump Table
;
iop:    jp      zero     ; Internal Status Routine (IOP+00h)
        jp      select   ; Device Select Routine   (IOP+03h)
        jp      zero     ; Device Name Routine     (IOP+06h)
        jp      iopinit  ; Initialize IOP          (IOP+09h)
;
cons:   jp      const    ; Console Input Status    (IOP+0Ch)
        jp      iconin   ; Console Input Character (IOP+0Fh)
        jp      conout   ; Console Output Character(IOP+12h)
        jp      list     ; List Output Character   (IOP+15h)
        jp      punch    ; Punch Output Character  (IOP+18h)
        jp      reader   ; Reader Input Character  (IOP+1Ah)
        jp      listst   ; List Output Status      (IOP+1Eh)
;
; Optional routines (not used by IOPCLK)
;
        jp      zero     ; I/O Driver Installation (IOP+21h)
        jp      zero     ; Open CON: Disk File     (IOP+24h)
        jp      zero     ; Close CON: Disk File    (IOP+27h)
        jp      zero     ; Open LST: Disk File     (IOP+2Ah)
        jp      zero     ; Close LST: Disk File    (IOP+2Dh)
;
;  I/O package identification
;
        db      'Z3IOP'  ; NZCOM IOP ID            (IOP+30h)
        db      'IOPCLK10' ; Package name (8 bytes)(IOP+35h)
;
; IOP Output Jump Table. ;                         (IOP+3Dh)
;
; The targets of the following seven jumps are filled in by
; the IOPINIT routine, always called by any IOP loader, to
; point to the NZBIOS jump table.
;
const:  jp      0
conin:  jp      0
conout: jp      0
list:   jp      0
punch:  jp      0
reader: jp      0
listst: jp      0

;=============================================================
;
; The preceding jumps and package ID MUST remain in their
; present positions to maintain the integrity of the IOP
; structure.
;
; The main body of the IO Package starts here. Code and
; structure from here on is free-form and may be moved
; about as you wish:
;
; IOPCLK configuration area.                    (IOP+52h)
;
; Configuration data is normally copied from the
; configuration area of the IOPCLK.COM loader program. If
; assembling this code to a REL file and using NZCOM or
; JETLDR to load it, configure this area directly in the
; source code:
;
beep:   db      0   ; ON=sound console bell as hourly alarm
seconds:db      0   ; ON=include seconds in clock display
time:   db      0   ; ON=use 12hr time
zsclk:  dw      0   ; Address of ZSDOS clock routine
;
; String sent to terminal to place a message at a specific;
; point in a terminal message field. The special character
; used to terminate the terminal message is located at
; TERMEND.  (0 terminations are for PSTR)
;
term:   ds      8   ; Prefix for terminal msg - 8 bytes max
        db      0   ; Termination
;
termend:db      0   ; Terminal message termination character
;
spaces: db      0   ; Number of spaces to the beginning of
                    ; the clock display
;
; The clock display buffer is a part of the terminal
; message. 'mo', 'da', etc, are filler bytes to show date
; and time locations. They are replaced with the current
; date and time bytes when the clock display is updated at
; UPDATM.
;
dmo:    db      'mo/'
dda:    db      'da/'
dyr:    db      'yr '
dhr:    db      'hr:'
dmi:    db      'mi'
colon:  db      ':'  ; Set to 0 to end with minutes
dse:    db      'se'
        db      0    ; Termination

;---------------
;
; Intercept CONIN
;
iconin: call    const       ; Check for pending character
        or      a
        jr      nz,conin    ; Yes, get it right away
;
        call    clock       ;
We have time, update clock display
        jr      iconin      ; Check for another character
;
clock:  ld      de,yr       ; DE=clock buffer
        ld      hl,(zsclk)  ; HL=address of ZSDOS clock driver
        ld      c,0         ; Set read clock function
        call    jphl        ; Read clock into our clock buffer
;
        ld      bc,se       ; Point to current seconds
        ld      hl,ose      ; Point to old seconds
;
chktim: call    secchk      ; Check for change (min or sec)
        ret     z           ; Quit if no change
;
; Update clock display buffer after initial check to see if
; seconds have changed, starting with minutes (or hours) and
; working backwards toward years. Convert each changed byte
; from BCD in the clock buffer to ASCII in the display
; buffer, quitting at the first unchanged date/time byte.
;
updatm: ld      de,dmi      ; Point to display buffer minutes
        call    check       ; Update if changed minutes
        jr      z,display   ; Quit if no change
;
updath: ld      de,dhr      ; Point to display buffer hour
        call    check       ; Update if changed hour
        jr      z,display   ; Quit if no change
;
        call    hour        ; Do change of hour housekeeping
        ld      de,dda      ; Point to display buffer day
        call    check       ; Update if changed day
        jr      z,display   ; Quit if no change
;
        ld      de,dmo      ; Point to display buffer month
```

requires a different terminal control string to identify such a message and the required string is not included in the ZCPR3 Z3TCAP, you have to do a little homework with your terminal manual to find the correct sequences.

On a WYSE30/50, for example, to place the clock display in the right corner of the host terminal message field, we need to send the following string (including the 28 spaces):

```
ESC,'F                            mm/dd/yy hh:mm:ss',CR
```

To do the same thing on a WYSE75, we need to send the following string (including the 22 spaces):

```
ESC,'[>+\                      mm/dd/yy hh:mm:ss\'
```

Reading the Clock

To keep things as universal as possible, I decided to use the ZDDOS/ZSDOS clock as my time and date source. Since the IOP clock is essentially a part of the BIOS, we can't take the easy way out and use the standard ZSDOS call to read the clock. While we could take advantage of ZSDOS support for re-entrant BDOS calls, this would require us to save and restore a 147 byte DOS data area, which would slow us down accordingly.

Luckily, there is an easier and faster way. Since ZSDOS stores the address of the ZSDOS clock driver at DOS+16h, we can easily find the address of the clock driver when the IOP clock is first loaded and save it in the IOP. We can then save time, trouble and IOP space by reading the ZSDOS clock directly through the clock driver. This sneaky technique also allows the IOP clock to continue working even if another DOS is later loaded, as long as the clock driver remains undisturbed.

IOP Clock Operation

CLKIOP (*Listing 1*), the IOP clock module, works by intercepting BIOS CONIN requests at ICONIN. We need to minimize the time we spend reading the clock and updating the clock display to avoid getting in the way of foreground application programs, so our first act is to check the console status to see if the BIOS already has a character waiting. If it does, we get it and return with it immediately. Only if there is no character waiting do we pause to read the clock and update the clock display. This allows the IOP clock to run in the background and take second place to applications utilities being run in the foreground.

Each time we update the clock display, we save a copy of the updated date/time buffer to help us minimize the time it takes us to update the display. We compare each newly-read date/time byte with the previously updated date/time byte as we step through the process of translating the BCD date and time bytes we read from the clock to the ASCII bytes we need in the clock display string buffer. Since seconds change most often, we can shorten the translation process by starting with seconds and working backwards toward years. As soon as we find an unchanged byte, we know the update is complete.

When we've completed updating our clock display buffer, we send the terminal control string to the terminal, which will direct it to place the following message in the message display field. Then we send the number of space characters required to position the clock display where we want it in the field, the clock display string, and the special message field termination character. Finally, we return to

```
        call    check     ; Update if changed month
        jr      z,display ; Quit if no change
;
        ld      de,dyr    ; Point to display buffer year
        call    check     ; Update if changed year,
                          ; else fall thru
;
; Update terminal clock display
;
display:ld      hl,term   ; Point to terminal message
                          ; line prefix
        call    pstr      ; Send it
;
        ld      a,(spaces) ; Number of spaces required
        or      a          ; before clock display
        jr      z,clkdisp  ; No spaces required
;
        ld      b,a       ; Set counter
        ld      a,' '
;
splp:   call    cout      ; Display spaces
        djnz    splp
;
clkdisp:ld      hl,dmo    ; Point to clock buffer
        call    pstr      ; Display it
;
        ld      a,(termend) ; Fall thru to send character

;----------------
;
; Subroutines:
;
; Display character in A on console
;
cout:   push    hl        ; Save us from the BIOS...
        push    bc
        ld      c,a       ; BIOS wants character in C
        call    conout    ; Display it
        pop     bc
        pop     hl
        ret
;
; Display 0-terminated string pointed to by HL
;
pstr:   ld      a,(hl)    ; Get character
        inc     hl        ; Point to next
        or      a         ; Done?
        ret     z         ; Yes, quit
;
        call    cout      ; Display character
        jr      pstr      ; Do next
;
; Call (HL)
;
jphl:   jp      (hl)
;
; Start with check for changed seconds
;
secchk: ld      de,dse    ; Point to display seconds
        jr      check0
;
; Start with check for changed minutes
;
minchk: ld      de,dmi    ; Point to display minutes
;
; Check for changed time/date bytes
;
check:  dec     hl        ; Point to next old time byte
        dec     bc        ; Point to next time byte
;
check0: ld      a,(bc)    ; Get byte
        cp      (hl)      ; Compare and set flags
        ret     z         ; Return if no change
;
        ld      (hl),a    ; Else update old buffer byte
        jp      ma2hc     ; And update display buffer
;
; Change of hour, so check if hourly alarm is desired
;
hour:   ld      a,(beep)  ; Hourly alarm?
```

```
        or      a
        ld      a,bell    ; Send beep if so
        call    nz,cout
;
; Format clock display correctly for 12 or 24 hour time
;
        ld      a,(time)  ; Get time flag
        or      a
        ret     z         ; 24hr, so we're done
;
        ld      a,(hr)    ; Get bcd hour
        sub     12h
        jr      z,noon    ; Noon
        jr      c,morn    ; Morning
;
        daa               ; Decimal adjust time to bcd
        jr      condun
;
morn:   add     a,12h
        jr      nz,condun ; Not Midnight
;
noon:   add     a,12h
condun: ld      de,dhr
        call    ma2hc     ; Convert display to 12hr
;
        ld      a,(dhr)   ; Check for leading 0
        cp      '0'
        ret     nz        ; No, we're done
;
        ld      a,' '     ; Replace leading 0 with a space
        ld      (dhr),a
        ret
;
; Device Select Routine
;
;  Entry: B=0FFH - Remove IOP
;  Exit : A=0, Z if bad command, A<>0, NZ if ok
;
select: inc     b
        jr      z,remove  ; Turn clock off (remove)
;
; Any IOP functions not implemented return ZERO to the
caller.
;
zero:   xor     a         ; Any call here returns Z
        ret
;
; Remove IOP
;
remove: ld      hl,cons   ; Our input jump table
        ld      de,const  ; Our output jump table
        ld      b,7       ; Patch 7 jumps
;
; Replace jump target at HL+1 with DE,
; increment HL and DE by 3.
;
target: inc     hl        ; Point to our jump target
        ld      (hl),e    ; Move low order
        inc     hl        ; Point to high order
        ld      (hl),d    ; Move it
        inc     hl        ; Point to next IOP input jump
        inc     de        ; Point to next IOP output jump
        inc     de
        inc     de
        djnz    target    ; Loop to do next
;
nzero:  or      0ffh      ; Any call here returns NZ
        ret
;
; IOPINIT is one-time initialization code that is always
; called by the IOP loader when the IOP is first loaded and
; never again. The space used by IOPINIT can be reused as
; buffer space if required.
;
; The seven character IO jumps in the NZBIOS jump table are
; vectored to our IOP Input Jump Table. The NZBIOS has a
; jump table similar to our IOP Output Jump Table whose
; targets go directly to the BIOS character IO routines. The
; address of this table is maintained at NZBIOS+1. The IOP
; is 'installed' by pointing the jumps in our IOP Output
```

ICONIN to check the BIOS for another character and if none is waiting, we continue to loop, reading and updating the clock display in our spare time, until a character is returned by the BIOS. We then return to the calling program with the character that was originally requested.

Loading the IOP Clock Module

You can customize CLKIOP's configuration buffer, including the terminal control string and the address for the ZSDOS clock driver, assemble the completed IOP clock module to a REL file and load it into the IOP buffer with NZCOM or JETLDR. It is just over 400 bytes long and fits nicely into a 'standard' 12 record (1.5k) NZIOP buffer with plenty of elbow room left over for future 'enhancements'.

Next time, however, we'll look at IOPLDR, a small general-purpose IOP loader REL module that does most of the standard work that it takes to load, control and remove an IOP module. We'll see how IOPLDR and a few routines specific to the IOP clock module can be combined to create IOPCLK.COM, a stand-alone utility to install and remove the IOP clock display module. IOPCLK also allows you to use ZCNFG to configure the operation of the final IOP clock module.

```
; Jump Table to the NZBIOS table.  If no IOP action is to
; take place, our IOP Input Jump Table vectors directly
; through the IOP Output Jump Table to the NZBIOS jump table.
;
iopinit:ld      a,(seconds) ; Include seconds?
        or      a
        jr      nz,init     ; Yes
;
        xor     a           ; Patch display to end
                            ; with minutes
        ld      (colon),a
        ld      hl,minchk   ; Patch addr of change
                            ; checking routine
        ld      (chktim+1),hl
;
        ld      a,0c3h      ; Don't recheck minutes
        ld      (updatm),a
        ld      hl,updath   ; Skip to checking hours
        ld      (updatm+1),hl
;
; Initialize IOP output jump table
;
init:   ld      hl,(1)      ; BIOS+3
        dec     hl          ; BIOS+2
        ld      d,(hl)
        dec     hl          ; BIOS+1
        ld      e,(hl)      ; BIOS table
        ld      hl,const    ; Our IOP output jump table
;
        ld      b,7         ; Seven jumps
        jr      target
;
; Clock buffers
;
yr:     db      0           ; Year
mo:     db      0           ; Month
da:     db      0           ; Day
hr:     db      0           ; Hour
mi:     db      0           ; Minute
se:     db      0           ; Second
;
; Copy of clock buffer after last display update. Fill
; character must be non-bcd character to ensure complete
; display conversion at startup.
;
oyr:    db      on          ; Last updated year
omo:    db      on          ; ...Month
oda:    db      on          ; ...Day
ohr:    db      on          ; ...Hour
omi:    db      on          ; ...Minute
ose:    db      on          ; ...Second
        end
;
; End of CLKIOP.Z80
```

Removing the IOPCLK IOP Module

An IOP module can be removed by simply loading another IOP module over top of it. Alternatively, IOPCLK.COM (when we have created it!) will be able to 'remove' the clock IOP module when the 'R' option is selected by calling the IOP SELECT routine with B=0FFh, causing the SELECT routine to restore the original IOP input jump table so that all jumps point directly to the output jump table. This produces a 'dummy' IOP module that simply passes on all BIOS calls without interception and effectively deselects the IOP.

Interaction With Applications Programs

Application programs usually get input characters from the user in one of two ways. When programs want to quickly check to see if the user has entered a character and continue with their work if nothing is waiting, they call the BIOS CONST routine. If no character is waiting, the program can then continue merrily on its way. If there is a waiting character, it can be retrieved by calling the BIOS CONIN routine. The SYSLIB character input routine CONDIN provides a simple method of doing this, returning immediately if no character is waiting and returning with the character if one has been entered.

The IOP clock works best with foreground applications utilities that get characters from the BIOS using BIOS CONIN requests. This method is typically used when a program can sit and wait until a character has been entered. The SYSLIB character input routine CIN works this way, waiting until a character has been entered and then returning with it. Most interactive utilities, including WordStar, ZPATCH, ZP, ZDB, IMP, SCAN24, et cetera, use CONIN calls for most character input. The IOP clock display is constantly updated in the background when these programs are operating.

Instead of calling CONIN and waiting for a character to be returned, however, some utilities try to do the job of the BIOS themselves, continually calling CONST until a character is detected and only requesting the character from CONIN after they already know that a character has been entered. On a call to CONIN, the IOP clock module sees that a character is waiting and won't stop to update the IOP clock display. As a result, the IOP clock module can't read the clock and update its display during operation of programs that get their character input this way. Some examples are dBase-II, ZDE, QL, XOX, VIEW and VTYPE.

Other utilities, including assemblers, linkers, and copy and directory utilities, don't check for console input at all for long periods of time. The IOP clock display will not be updated when console input is not being requested. The display will be updated again, however, at the next console input request when there is no character already waiting.

Why then, you might ask, don't we just make life easier by intercepting CONST calls to read the clock and update the IOP clock display? The reason is that CONST is called much more often than CONIN and it just slows things down too much to read the clock that often. Or at least my Ampro BIOS clock, which is pretty fast. Reading the clock during each CONST call drastically reduces the apparent terminal

Boyer-Moore String Search Algorithm

By Terry Hazen

Bridger Mitchell's last *Advanced CP/M* column (*TCJ* 45) described a Z80 implementation of the Boyer-Moore string search algorithm. He illustrated the traditional method of creating a displacement table to determine the skip values required during the search process. However, there is a simpler and faster way. Since at each node we try to match the pattern to the text starting with the rightmost pattern character and working from right to left, we can derive each skip value as a by-product of the search process itself.

For example, if we load A with the text character we are searching for in the pattern, HL with the pointer to the end of the pattern and BC with the length of the pattern, we can search the pattern from right to left with the Z80 instruction CPDR. If CPDR returns a NZ flag, we have no match and must use a skip value equal to the length of the pattern. If we have a match, BC returns with an offset from the beginning, or left character, of the pattern to the match position. The skip value for the next match attempt is simply the offset of the match position from the right end of the pattern, or:

```
        skip = pattern_length - BC - 1
```

Using one of Bridger's examples:

```
            1 2 3 4 5 6 7 8 9 0
Text:       e v e r y w h e r e
Pattern:        w h e r e
                        ^
```

Here, A would contain 'h', HL would point to the right hand 'e' in the pattern 'where', and BC would contain 5, the length of the pattern. The CPDR instruction will quit when it matches the 'h' in 'where' and return BC=1. The next skip value would then be 5-1-1=3:

```
            1 2 3 4 5 6 7 8 9 0
Text:       e v e r y w h e r e
Pattern:        w h e r e     |
                - - > w h e r e    ; Slide pattern right 3 bytes
                              ^
```

While the use of the CPDR instruction illustrates how the search process itself can generate the skip value for the next match attempt, its actual usefulness in matching real text characters in the pattern is limited, as you are limited to exact case matches. More flexible 'discrete' code that allows you to match characters ignoring case and allowing wildcard pattern characters is more complicated and slightly slower, but does the same job. ●

baud rate and slows down screen display updates dramatically and unacceptably.

Bells and Whistles

To avoid obscuring the main points, I didn't discuss all of the features you will find in the CLKIOP clock display code. For example, you can use configure the IOP clock module to sound an hourly alarm. It can also be configured to display time as hrs:mins with the display updated every minute, or as hrs:mins:secs with its display updated every second and can be set for 12 or 24 hour time display.

You can find the full source and documentation for CLKIOP and the IOP clock display utility, IOPCLK.COM, in IOPCLK10.LBR on your favorite Z-Node. I hope IOPCLK helps illustrate another way in which an IOP can be used even if you can't use the IOP clock display because you don't have a clock, your terminal doesn't have a host terminal message field or you use a different BDOS. Maybe you have some ideas of your own for new IOP's.

Next time we'll look at IOPLDR and use it to create the stand-alone IOP clock display utility IOPCLK.COM. ●

Promotional and Technical Writing for Electronics Marketing

★ ★ ★ ★ ★

Technical Articles for Publication
Advertising Concepts and Copy
Product and Service Brochures
Press Releases
Speeches and Lectures
Editing/Rewrite Service
Consulting

★ ★ ★ ★ ★

Bruce Morgen
P.O. Box 2781
Warminster, PA 18974
215-443-9031
(Voice, Data by Appointment)

Servos and the F68HC11

By Matt Mercaldo

The servo is a wonderful device that can be used for motion control. These small units have high turning torque as well as high holding torque. A limitation of servos is that most do not turn a full three hundred and sixty degrees. In most small robotic applications this is not a factor. The following code and article will describe how to turn a servo. The servo used is a Futaba S9601. I have to thank the manager of a little hobby shop nearby for the donation of this unit from his "junk box".

Onward

Servos work in a curious manner in that a control signal determines what position should be taken next. This signal is a regular sixteen millisecond wave whose duty cycle determines the servo's position. The position is determined from a one to two millisecond window or duty cycle. *Listing 1* is the code for the servo. The constant BIAS is an offset from the start of the sixteen millisecond wave cycle. The constant STEP tells how many system E clock ticks for a "step".

The wave in *figure 1* is what the signal looks like to run a servo. From point 1. to point 4. is sixteen milliseconds. Point 1. to point 2. is the BIAS. Point 2. to point 3. is the window of movement. In the following code, this section (point 2 to point 3) is divided into 256 steps.

A look at the Code

The timers in a 68HC11 are based on the two megahertz E clock. A full 65535 ticks is 32 milliseconds. 32720 ticks are roughly half of that and constitute a 16 millisecond FRAME in the code. The wave table is generated by MAKE_STEP_TABLE. The table has 256 elements. Each element is composed of two, sixteen bit numbers. The first number is a count of how many ticks of the E clock make up the high part of the wave while the second number is a count of how many ticks of the E clock make up the low part of the wave. The timer interrupt service routine is ~SERVO. NEXT_SERVO is a pointer to a piece of code that knows what part of the wave to build next. There are only two parts to the wave that can be built; the up part and the down part. ~SERVO_UP sets the signal line bit high then gets the appropriate count from a combination of the position table and the POSITION offset into the table, and finally sets the NEXT_SERVO vector to point to ~SERVO_DN. The only difference between ~SERVO_UP and ~SERVO_DN is that ~SERVO_UP accesses the up part of the wave table element and sets

```
( Primitive Servo Controller      )
( By Matthew Mercaldo             )
(    41 South Park Street         )
(    Oconomowoc, Wisconsin  53066 )

( Servo used is the FUTABA S9601 mini servo )
( This module will establish 256 locations for the
( revolution of the servo )
( Down Load the Startup code and 68HC11 assembler before )
( downloading this module )

HEX
B000 CONSTANT PORTA

DECIMAL
OC1F CONSTANT SIGNAL_BIT   ( PORT A bit 7 is the signal bit )
( connect the white lead of the servo to PORTA bit 7, the
( black lead to ground and the red lead to 5 volts )

768  CONSTANT BIAS            ( Bias is the amount of time     )
                              ( required to not make the servo )
                              ( chatter at the low positions.  )
15   CONSTANT STEP            ( There will be 13 system E clock)
                              ( ticks per "step" of servo.     )
: PS ( ps - count )   STEP * BIAS + ;
: FRAME ( - count )   32720 ;  ( 16 milliseconds )
: MAKE_STEP_TABLE ( - ; compiles table into the dictionary  )
                  (    ; the table is 16bit UP | 16bit DOWN  )
                  (    ; for 256 count                       )
     256 0 DO   I PS ,  FRAME I PS - ,  LOOP
;

( create the wave/ position table )
CREATE POSITION_TABLE      MAKE_STEP_TABLE

VARIABLE NEXT_SERVO  ( A vector which points to the next )
                     ( routine to fire )
VARIABLE POSITION    ( offset to the waveform pair )
VARIABLE ~^SERVO_UP  ( used for defered access to the up and )
                     ( down routines )
VARIABLE ~^SERVO_DN
```

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!)

```
CODE EI  ( - ;enable all interrupts )
ASSEMBLER
     CLI   NEXT ^ JMP
END-CODE

CODE DI  ( - ;disable all interrupts )
ASSEMBLER
     SEI   NEXT ^ JMP
END-CODE

CREATE -SERVO  ( - ; Servo interrupt routine )
ASSEMBLER
     OC1F # A LDA      TFLG1 ^ A STA  ( Acknowledge Interrupt )
     NEXT_SERVO ^ LDX   0 ,X JMP      ( execute appropriate   )
                                      ( routine )

CREATE -SERVO_UP ( - ; makes bit go high at appropriate time)
ASSEMBLER
     PORTA SIGNAL_BIT ON                      ( toggle bit on )
     POSITION ^ LDD    POSITION_TABLE # ADDD  XGDX
                                              ( set timer for )

     0 ,X LDD   TCNT ^ ADDD    TOC1 ^ STD     ( next time )
     -^SERVO_DN ^ LDX  NEXT_SERVO ^ STX       ( set vector for)
                                              ( next time )
     RTI

-SERVO_UP -^SERVO_UP !

CREATE -SERVO_DN  ( - ; makes bit go low at appropriate time)
ASSEMBLER
     PORTA SIGNAL_BIT OFF                    ( toggle bit off )
     POSITION ^ LDD    POSITION_TABLE # ADDD  XGDX
     2 ,X LDD   TCNT ^ ADDD    TOC1 ^ STD
     -^SERVO_UP ^ LDX  NEXT_SERVO ^ STX
     RTI

-SERVO_DN -^SERVO_DN !


 : POSITION_SET ( n - ; set position to be an offset into   )
                                             ( the wave table )
      DUP 255 > IF   DROP 255  THEN
      4 *  POSITION !
 ;
 HEX

 : SERVO_INIT ( - ;set servo timer up )
     00 PORTA C!    ( oc1 pin starts low )
     80 B026 C!
     00 B020 C!
     -SERVO >TOC1 VECTOR
     -SERVO_UP NEXT_SERVO !
     0 POSITION_SET
 ;

: SERVO_START ( p - )
     DI
     SERVO_INIT
     TMSK1 C@ OC1F OR TMSK1 C!
     EI
;
```

Figure 1

```
                 _______________                 _____
    ____/          \\\\\\\\\\\\________________/

         |---------|-----------|---------------|
         1         2           3               4
```

the signal bit high while ~SERVO_DN accesses the down part of the wave table element and sets the signal bit low. Together ~SERVO_UP and ~SERVO_DN create the servo control signal. The POSITION is set with the word POSITION_SET. Since each element in the table is four bytes long, the entered number is multiplied by four. The reason why each element in the wave table is four bytes long is because the 68HC11 counters are sixteen bits or two bytes, and there are two of them. SERVO_START turns off interrupts, initializes the timers and port A for its use as the signal port. Port A bit 7 is the signal output line. The servo will go to position 0 once activated. To move the servo first type SERVO_START <CR> to initialize the servo timers as described above, then use the POSITION_SET command by typing (for example) 50 POSITION_SET <CR>.

Servos are a tight, compact, powerful little actuators. In future articles we will use them to build wondrous robotic toys. I hope you have as much fun experimenting with servos as I did in researching this article. Once again have fun with Forth!●

Baseball provides an escape. Furthermore, there is no other place in society that I know of which the perimeter of play and the rules are clearly defined and known to everyone—in which justice is absolutely equal and sure. Three strikes, you're out. I don't care if you hire Edward Bennet Williams to defend you; three strikes, you're still out. Baseball is an island of stability on an unstable world.

—Bill Veeck

Z-System Corner

Programming for Compatibility: Z-System and CP/M

By Jay Sage

As noted in the sidebar to my columns, I serve as the Z-System sysop on GEnie. My principal duty is to conduct a Real Time Conference (RTC) at 10 pm Eastern time on the first Wednesday of each month. These sessions are group chats, and they give GEnie callers a chance to ask questions and make comments. When I accepted this position with GEnie, I had expected that many Z-System enthusiasts would take advantage of the opportunity to discuss Z-System issues with me, but it has not turned out that way. Relatively few people show up for these sessions.

They have not been without value, however, and one of the most valuable took place this past June. David Goodenough, the author of QTERM (the program for Z80 computers that is currently the number-one subject of interest), was in attendance as usual, and he and I got into a discussion about the lack of sharing and commonality between the Z-System community and the community of vanilla CP/M users.

The Issue of Compatibility

The discussion started when I mentioned that we were beginning work on a replacement for BYE. David asked if it would work under CP/M, and I replied that it would absolutely require a Z-System. David found this very annoying and complained about the way the Z community ignores the large part of the 8-bit computing world that, for whatever reason, sticks with standard CP/M.

In the case of BYE, I do not accept David's criticism. The main motivation for writing a new version of BYE is that many of the functions and a large part of the code in BYE is completely unnecessary on a Z-System. For example, Z-Systems are inherently secure and do not need BYE to provide security. Furthermore, the externally accessible multiple command line makes it possible for BYE to have the operating system perform functions that are currently included in the BYE code.

A BYE for Z-Systems could be significantly smaller and more flexible than a BYE for CP/M, and we would not want to carry the overhead demanded by CP/M. David's response to that was to question, then, why he should carry any overhead for Z-System in the programs he writes (since he never uses any form of Z-System).

I think the difference is that BYE is a piece of resident code, something that becomes part of the operating system of the computer. As such, it is much more important for the code to be as short as possible. In an application or utility program, however, the importance of keeping code short is much less, and the importance of compatibility and wide-ranging applicability is much greater. With respect to those kinds of programs I think that David is absolutely right, and we should be making a greater effort at compatibility. That is the issue I will address here.

Compatibility, like motherhood, is almost good by definition!

Why Should We Want Compatibility?

Compatibility, like motherhood, is almost good by definition! Obviously, a great deal of creative effort is going into Z-System development, and it is best if the fruits of that effort can be shared by the largest number of people. But there are also some very specific reasons why we in the Z community should be interested in compatibility with CP/M.

One reason that affects Z users directly is that with NZCOM and Z3PLUS we are not locked into Z-System, and we are much more likely to drop back to CP/M to perform some tasks. Sometimes we are going to use an application that is a real memory hog. At other times we have to use a program that cannot operate under Z-System, such as Uniform (for working with foreign-format diskettes). Uniform works by introducing patches directly into the operating system code, and it will work, therefore, only when the machine is running the exact version of CP/M for which it was written.

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.

(I have made perfunctory attempts at finding a way to adapt Uniform to NZCOM, but so far I have not succeeded.) It would be very convenient if our familiar Z-System tools would still work after we dropped back to CP/M. And we certainly don't want them causing a system crash if we invoke them by accident under CP/M.

A second advantage, and one that David Goodenough raised to encourage me to pursue greater compatibility, is that people still using vanilla CP/M would get a taste of what Z-System can offer. Under CP/M, Z-System tools would require an onerous patching process. If more CP/M

Listing 1.
Source code for the test program that incorporates an internal ENV/TCAP for when the program is run under standard CP/M.

```
; PROGRAM:        ZTEST.Z80
; AUTHOR:         Jay Sage
; DATE:           June 16, 1991

; This is a program to test the technique of including
; an internal ENV and TCAP.

; ---------- External References

        maclib          cpmenv.lib

        .request        vlib,syslib0

        extrn           z3vinit,cls,at  ; VLIB routines
        extrn           vprint

; ---------- Standard Z-System program header

; To make the code work transparently with CP/M, the
header
; is initialized to point to the internal ENV.  When the
; program is run under ZCPR33 or later, the address of the
; external Z-System ENV will be poked into the code by the
; command processor at run time.

        jp      start

        db      'Z3ENV'         ; Signature
        db      1               ; ENV type
envptr::
        dw      intenv          ; Pointer to ENV

; Material for use with ZCNFG can be included here.

; ---------- Internal ENV and TCAP placed here

intenv: cpmenv                  ; Use macro

; ---------- Actual Program Code

start:
        ld      hl,(envptr)     ; Initialize
        call    z3vinit

        call    cls             ; Clear screen
        call    at              ; Position cursor
        db      10,10
        call    vprint          ; Display a message
        db      'This is '
        db      1               ; Highlighting on
        db      'highlighted'
        db      2               ; Highlighting off
        db      ' video!'
        db      0
        call    at              ; Put cursor at bottom
        db      22,1

        ret
        end
```

Listing 2.
This is macro code that defines an ENV and TCAP for a CP/M system.

```
; PROGRAM:        CPMENV.LIB
; AUTHOR:         Jay Sage
; DATE:           June 16, 1991

; --------------------

; System configuration information (***** USER EDIT *****)

cpumhz  equ     4                ; CPU speed in MHz

; Operating system addresses and sizes.

biospg  equ     0d1h             ; Page where BIOS starts

bios    equ     100h * biospg
doss    equ     28               ; Size of DOS in records
dos     equ     bios - 80h * doss
ccps    equ     16               ; Size of CCP in records
ccp     equ     dos - 80h * ccps

; Information about drives and user areas available

;               PONMLKJIHGFEDCBA
drvec   equ     0000000000001111B
highdsk equ     'D'              ; Letter of highest drive
maxdisk equ     highdsk - '@'    ; Highest drive (A=1)
maxuser equ     31               ; Highest user area

; Data about console screen and printers

crtwid  equ     80               ; Width of CRT screen
crtlen  equ     24               ; Number of lines on
screen
crtuse  equ     crtlen -2        ; Number of lines to use

prtwid  equ     80               ; Printer width
prtlen  equ     66               ; Printer total length
prtuse  equ     prtlen - 8       ; Printer lines to use
prtff   equ     1                ; Formfeed flag (1 if
used)

; --------------------

; Here is a macro to define and internal ENV for use
; under CP/M.

cpmenv  macro

        jp      0       ; Dummy jump address

        db      'Z3ENV' ; Environment ID
        db      81h     ; ENV type

        dw      0       ; external path
        db      0       ; elements in path

        dw      0       ; RCP address
        db      0       ; number of records in RCP

        ; LOTS MORE OMITTED

        dw      intenv  ; ZCPR3 Environment Descriptor
        db      2       ; number of records in ENV

        ; LOTS MORE ZERO VALUES

        db      cpumhz  ; Processor Speed in MHz

        db      maxdisk ; maximum disk
        db      maxuser ; maximum user

        db      1       ; 1=OK to accept DU, 0=not OK

        db      0,0

        db      crtwid  ; width of CRT
```

users realized the kinds of programs that would become available to them directly, with no need for patching, they would be more likely to upgrade to Z-System.

What Does Compatibility Demand?

What do we mean when we talk about Z-System programs being compatible with CP/M, and what coding requirements would such compatibility impose?

As Bridger Mitchell pointed out long ago in his *Advanced CP/M* column that used to appear in *TCJ* (and which we all hope will reappear in the future [Ed.: I second that!]), all programs should examine the environment in which they have been called on to operate. Then, they must either adapt properly to the environment or terminate gracefully with an appropriate message.

All programs should examine the environment in which they have been called on to operate. Then, they must either adapt properly to the environment or terminate gracefully.

Two basic things that programs might have to determine before they attempt to perform their function are: (1) the kind of CPU—8080/8085, Z80, 64180, Z280; and (2) the kind of operating system—CP/M-2.2, CP/M-Plus, Z-System. In the case of a Z-System, it might be further necessary to determine the version of ZCPR3 (3.0, 3.3, 3.4) and the type of implementation (manual, NZCOM, Z3PLUS).

The Z-System is a far more varied object than CP/M, and one's determination of the environment has only begun when a Z-System has been detected. The simplest form of Z-System, I suppose, would have only one module: the ENV descriptor. The other modules and facilities in Z-System's long list are, I believe, all optional: named directory register (NDR), multiple command line, shell stack, flow control package (FCP), message buffer, command search path, and so on.

Because of the range of Z-System implementations that are possible, it is already incumbent on Z programs to make sure that the facilities they expect or depend on are, in fact, available. Most of the routines in the assembly language programming libraries used to write Z programs (i.e., VLIB, Z3LIB, and SYSLIB) already return error codes when the facility requested is not available. All Z programs should check these return codes and proceed in appropriate ways when things don't work out as intended.

What should be expect from Z programs when they are run under CP/M? At the very least, as I mentioned earlier, Z programs should absolutely be safe under CP/M. We might settle for a simple return to the CP/M command prompt, but ideally, programs should report that they could not run and why. It is quite unnerving to invoke a program and just get the command prompt back.

A higher level of compatibility would be for the Z program to perform those of its functions that make sense under CP/M. When running under Z-System the program might recognize named directories and return information in Z-System message buffer registers; under CP/M, only DU: directory references could be handled, and information that would normally be recorded in the message buffer would be discarded.

The highest level of compatibility, which we will explore in more detail later, is to have the program perform its full range of operations under CP/M by including an internal ENV module (including the TCAP component, which tells how to operate the console terminal). In this way, the program would think it was running under Z-System even when actually running under CP/M.

I have not had time to think through all these issues fully, and my views may be refined with further thought and input from others. At the moment, I can see only one fundamental difference between a minimum capability Z-System and a CP/M system. That is the difference in command processors, and specifically the more sophisticated parsing of the command line that ZCPR3 performs.

The first two tokens on the command line are treated as file references, and the default file control blocks (FCBs) at 5CH and 6CH are filled in with information about those two files. With ZCPR3, directory references of the form DU: and, if named directories are implemented, DIR: are recognized. Besides placing the proper drive letter into the FCB, the user number for the file is placed into a special location.

Under CP/M, these operations will not take place, and an internal ENV will not help. In fact, the internal ENV, which will make the program think that it is running under Z-System, could cause problems. To make programs that rely on the default FCB data work, we would have to add significant additional code to the program. First, we would have to detect the use of the internal ENV (this would be easy), and then we would have to provide alternative code for manually parsing the filename tokens into the FCBs. While this might represent a significant extra burden in the code, I would recommend that we do this in the future for those programs that can afford the extra code and whose functionality is not already available in a standard CP/M program.

What should be expect from Z programs when they are run under CP/M? At the very least, they should absolutely be safe.

Where Does Compatibility Stand Now?

David Goodenough was under the impression Z-System programmers totally ignore CP/M and that nearly all Z-System programs will not work under CP/M. This impression is not really fair.

First of all, a great many Z programs—perhaps even the majority—could not possibly run under CP/M because their function would make no sense under CP/M. Here are a few examples:

(1) PATH or ZPATH: configures the Z-System search path for COM files;

(2) PWD: reports the names and associated drive/users of currently defined named directories;

(3) SALIAS: full-screen tool for defining stand-alone multiple-command-line aliases;

(4) ZEX: a sophisticated batch processor that feeds commands to the multiple command line and uses the message buffer for control communication;

(5) ADIR: displays the names of alias scripts defined for the ARUNZ extended command processor;

(6) LSH: a full-screen history shell and command-line editor.

It was interesting to see what happened when these programs were operated under CP/M (which I could do easily

```
        db      crtlen  ; number of lines on CRT
        db      crtuse  ; number of lines of text on CRT

        dw      drvec
        db      0

        db      prtwid  ; data for printer
        db      prtlen
        db      prtuse
        db      prtff

        db      0,0,0,0

        dw      ccp
        db      ccps

        dw      dos
        db      doss

        dw      bios

        db      'SH      '      ; shell variable filename
        db      'VAR'           ; shell variable filetype

        db      '        '      ; filename 1
        db      '   '           ; filetype 1

        ; MORE SIMILAR DATA

;  Fill unused space with nulls

        rept    128-($-intenv)
        db      0
        endm

;  End of Environment Descriptor — beginning of TCAP

; ***** USER EDIT *****

; Extended Termcap Data

ESC     EQU     27      ; ASCII escape character

; I have adopted the convention that a terminal name is
; terminated with a space character, therefore no spaces
; within the name. Also that the terminal name is unique
; in the first eight characters.

NZTCAP: DB      'WYSE-50D     ' ; Terminal name (13 bytes)

; The Graphics section is no longer fixed so we must
; provide an offset to it.  One byte is sufficient for a
; two-record TCAP.

        DB      GOELD-NZTCAP    ; Offset to GOELD

; Bit 7 of B14 indicates the new Extended TCAP.  Bits 6-0
; are undefined.

        DB      10000000B       ; Extended TCAP

; B15 b0        Standout        0 = dim, 1 = inverse
; B15 b1        Power Up Delay  0 = None, 1 = 10-sec delay
; B15 b2        No Wrap         0 = Line Wrap, 1 = No Wrap
; B15 b3        No Scroll       0 = Scroll, 1 = No Scroll
; B15 b4        ANSI            0 = ASCII, 1 = ANSI

        DB      00000111B

        DB      'K'-'@'         ; Cursor up
        DB      'J'-'@'         ; Cursor down
        DB      'L'-'@'         ; Cursor right
        DB      'H'-'@'         ; Cursor left

        DB      00              ; Clear-screen delay
        DB      00              ; Cursor movement delay
        DB      00              ; Clear-to-end-of-line
delay

; Strings start here.
```

on my Televideo 803H, which runs NZCOM). ZPATH politely reported that there was no ZCPR3 path, and PWD announced that there was no NDR (named directory register) allocated. ZEX gave a more general message indicating that the facilities it required for operation were not available. These responses were all acceptable and reasonable, and they meet the requirements I outlined above.

ADIR did not do so well. It tried to run and ended up accessing a bogus drive, from which I had to recover by pressing control-C. SALIAS, to my surprise, did even worse. Although one time it gave me the message "TCAP?", indicating that it was checking the TCAP for adequate terminal support, all the other times it crashed and locked up the system. So did LSH. Obviously, these programs are not checking properly that the memory referenced by the embedded ENV

```
        DB      ESC,'+',0       ; Clear-screen string
        DB      ESC,'=%+ %+ ',0 ; Cursor movement string
        DB      ESC,'T',0       ; Clear-to-end-of-line
        DB      ESC,')',0       ; Standout-on string
        DB      ESC,'(',0       ; Standout-end string
        DB      0               ; Terminal init string
        DB      ESC,'(',0       ; Terminal deinit string

; Extensions to Standard Z3TCAP

        DB      ESC,'R',0       ; Line Delete
        DB      ESC,'E',0       ; Line Insert
        DB      ESC,'Y',0       ; Clear-to-end-of-screen

; Set Attribute strings once again included.

        DB      ESC,'G',0       ; Set Attributes
        DB      '0248',0        ; Attributes

; These two allow reading the Terminal's screen.

        DB      ESC,'?',0       ; Read current cursor pos
        DB      ESC,'6',0       ; Read line until cursor

; Graphics start here.

GOELD:  DB      0               ; On/Off Delay

; Graphics strings offset from Delay value.

        DB      ESC,'H',2,0     ; Graphics mode On
        DB      ESC,'H',3,0     ; Graphics mode Off
        DB      ESC,''0',0      ; Cursor Off
        DB      ESC,''1',0      ; Cursor On

; Graphics Characters

        DB      '2'             ; Upper left corner
        DB      '3'             ; Upper right corner
        DB      '1'             ; Lower left corner
        DB      '5'             ; Lower right corner
        DB      ':'             ; Horizontal line
        DB      '6'             ; Vertical line
        DB      '7'             ; Full block
        DB      ';'             ; Hashed block
        DB      '0'             ; Upper intersect
        DB      '='             ; Lower intersect
        DB      '8'             ; Mid intersect
        DB      '9'             ; Right intersect
        DB      '4'             ; Left intersect

;  Fill unused space with nulls

        REPT    128-($-NZTCAP)
        DB      0
        ENDM

;  End of NZTCAPD

        endm
```

pointer actually contains a Z-System environment (ENV) module; they must just be plowing on ahead.

I tried a lot of other programs of this sort, and in almost all cases they failed in an acceptable way. Many gave messages; others simply returned to the command prompt without having done anything (at least nothing apparent, so I assume they did not do any harm).

Not all Z-System programs manipulate or take specific advantage of Z-System facilities; There are many Z programs that perform functions that make perfect sense under standard CP/M. Some authors have taken great care to ensure that their programs will work in both environments. Hal Bower's COPY program (derived from MCOPY) that comes with the ZSDOS disk-operating-system replacement is a good example of this. Here are some other examples of programs whose function makes some sense under CP/M:

(1) FF: FindFile searches all drives and user areas for specified file names;

(2) DIFF: performs a byte-by-byte comparison of two files;

(3) CD: changes the logged drive/user;

(4) LBREXT: extracts member files from LBR files and optionally uncompresses them;

(5) LPUT: builds a new LBR file from the files named on the command line.

How did these fare under CP/M? Although FF performs a function that would be useful under standard CP/M, it is coded to make mandatory use of Z-System features, and it delivers an error message when one attempts to use it under CP/M.

This is a good example of a program that probably should be upgraded to CP/M compatibility. Obviously, FF would not recognize or report named directories, and it would have to assume that all user areas are available. As for the drives it should search, it would have to be configured manually, and I don't see why ZCNFG, the program that is currently used under Z-System to configure it, could not perform this function under CP/M, too.

DIFF, like FF, gives an error message when it is invoked under CP/M (in fact, it requires ZCPR33 or later). DIFF makes extensive use of Z-System facilities. It determines the dimensions of the terminal display, stores certain results of the file comparison in Z-System registers, and performs advanced error-recovery operations when the program encounters certain conditions. DIFF also extracts and compares the date stamps for the two files. Thus, although the essential function of DIFF would be useful under CP/M, it might be difficult to make the code work without those Z facilities. It would be worth looking into, I think.

CD (Change Directory) is primarily intended to log into another drive and user area using a named-directory reference and to run a special alias when it gets there. Under CP/M it currently appears to do nothing but return to the command prompt. It should be made to give an error message. It might even be reasonable for it to accept DU: syntax to log into the indicated area under CP/M.

The functions of LBREXT are totally appropriate for CP/M, and there is no reason why it should not work perfectly under CP/M. It almost does. The following command works fine:

```
LBREXT B3:LIBRARY C1:EXTRACT.FIL
```

As is perhaps to be expected, it does have problems if one tries to use a named directory reference. The one real mistake I noticed in the code is that it is coded to determine its own name (for use in the help screen) by looking in the Z-System external FCB. Under CP/M, garbage appears. Apparently, the code does not verify that the system has an external FCB.

Listing 3.
The essential material in the patch for installing an internal environment into DU35.

```
; PROGRAM:      DU-CPM.Z80
; AUTHOR:       Jay Sage
; DATE:         June 15, 1991

; This code is a patch that can be used with DU35 to embed
; an internal environment descriptor and TCAP so that DU35
; can be used under standard CP/M (2.2 or 3) as well as Z-
; -System. To use this patch, DU35.Z80 must be assembled as
; usual to a REL file and then linked with the data segment
; (DSEG) moved 100H higher in memory to provide a place to
; insert the ENV. The appropriate linker commands with the
; version-4 libraries are as follows:
;
;       SLRNK DU35/N,/P:100,/D:2E50,DU35,
;                     VLIB/S,Z3LIB/S,SYSLIB0/S,/E
; or
;       ZML DU35,VLIB/,Z3LIB/,SYSLIB0/ D2E50
;
; This linking leaves the space from 2D50H to 2E4FH free for
; an internal ENV and TCAP, which are defined below in this
; patch. As distributed, this patch is set up with an ENV
; for my Televideo 803H computer and a TCAP for a Wyse-50
; terminal (this works on the Televideo, too). You should
; edit the file so that it describes your system (search for
; the sections marked with "***** USER EDIT *****".
;
; Once the patch has been edited, assemble it to a HEX file
; and then use MLOAD (or MYLOAD) to apply the overlay:
;
;       MLOAD DU=DU35.COM,DU-CPM.HEX
```

```
; An alternative way to install the TCAP is to extract a
; binary TCAP from one of the distribution files (such as
; Z3TCAP.TCP, which is a library collection of TCAPs)
; Then, after installing the DU-CPM patch, use a debugger or
; file editor to install the desired TCAP into DU.COM at
; address 2DD0H (on top of the one installed by this patch).

; ------------------------------------------------------------

; System configuration information (***** USER EDIT *****)

        ; OMITTED MATERIAL SAME AS THE CONFIGURATION
        ; EQUATES IN CPMENV.LIB

intenv  equ     02d50h  ; Place in DU35 for the internal ENV

; ------------------------------------------------------------

; Install ENV address at beginning of code

        org     109h            ; Place for the ENV address

        dw      intenv          ; Internal ENV address

; ------------------------------------------------------------

; Install the dummy ENV in the DU35 code

        org     intenv

        ; OMITTED MATERIAL SAME AS THE MACRO CODE
        ; IN CPMENV.LIB

;  End of NZTCAPD
```

This is a mistake even for operation under Z-System.

LPUT, which also performs a function that is entirely appropriate for CP/M, misbehaves seriously under CP/M. It tries to work, but it gets its user areas confused. Since it always assumes user 0 for the LBR file, even when one is given explicitly in the command, my guess is that it is using the default FCB for the first command line token. Second, for the list of files to be put into the library, LPUT assumes user 0 unless one is given explicitly, in which case it is recognized correctly. This suggests that it has not determined properly what its logged-in (default) user area is. Small changes in the code could probably correct these problems.

I probably should have clarified one thing earlier in this discussion. I have not done these experiments strictly under CP/M but rather under ZCPR2, and this may have affected the results. A stripped-down ZCPR2 is the most primitive system I will consider running. It is a direct, drop-in replacement for the Digital Research command processor; nothing else in the system changes. I can imagine (albeit with some difficulty) why someone would still today refuse to use Z-System (it does take up some memory), but it is completely beyond me as to why anyone would continue to use the DRI CP/M-2.2 CCP.

I can imagine why someone would still today refuse to use Z-System (it does take up some memory), but it is completely beyond me as to why anyone would continue to use the DRI CP/M-2.2 CCP.

Compatibility Via an Internal Environment

I was so intrigued by the possibility of making Z programs run under CP/M even when they required full-screen terminal capabilities that I could not wait to experiment with some real code. Mind you, this is no small thing. I have been so busy with other activities that I have done virtually no code writing for years! My influence on the Z community has been only as a mentor. This time I just could not wait for someone else to perform the experiments.

As the subject for my first tests, I chose the disk utility program, DU3. There were two reasons for this. First, this is a program whose CP/M counterpart lacks some of the most useful features of DU3, and I have long wished that it would work after I dropped down to CP/M. Second, I had just put up on my Z-Node a new version, DU34, that Gorm Helt-Hansen of Denmark had sent to me, so I knew I had current source code to work with.

There are quite a few possible approaches to patching in an internal ENV. I am going to start by discussing a little test program that I wrote after I had already succeeded with DU34 (now DU35). Rather than trying to abstract from the DU code, it was easier to write a simple demonstration program. ZTEST, shown in *Listing 1*, includes only enough functionality to prove that it works.

The key idea is to place in the code a two-record module containing the ENV (1 record) and the TCAP (1 record). The ENV pointer in the program header is initialized to point to this internal environment. Then, when the program is executed under CP/M, that is the ENV that the program will see and use. On the other hand, when the program is run under a modern Z-System (ZCPR33 or later), the command processor will poke the address of the true (external) Z-System ENV into that pointer, and the program will then see and use the real environment.

The most difficult part of this project was writing the CPMENV.LIB code. The ENV part was pretty straightforward. Almost all the module addresses and lengths are zero! The part that took some time was the TCAP. I finally located a good source-code version of the latest TCAP standard and was able to import it. A condensed version is shown in Listing 2.

For patching DU3, I took a slightly simpler approach that would not have required modifying the source code at all. In fact, I did make a few changes to correct some errors I noticed in the version 3.4 code and to make a cosmetic change. For one thing, the names of some library routines had mistakenly shorted to six characters, rendering the code unusable with the SLR tools in SLR mode. Using the full names should be equally acceptable to M80/L80.

The most serious error was the inclusion of a large block of initialized data near the end of the data segment (DSEG). This resulted in a COM file substantially larger than necessary, since all the uninitialized data now had to be loaded into the COM file. I moved that initialized data to the beginning of the DSEG to join the other initialized data. I called the new version DU35.

Aside from those modifications, which were made for other reasons, the patching process actually starts with the same assembled REL file as would have been used to generate the standard Z-System version of the program. When I linked it with the libraries (VLIB, Z3LIB, and SYSLIB), I made note of where the data segment (DSEG) started. It was at 2D48H. To make room for an internal ENV from 2D50H to 2E4FH, I simply relinked the program with the DSEG specified as 2E50H. Now all I had to do was to patch in the ENV code.

A stripped-down ZCPR2 is the most primitive system I will consider running. It is a direct, drop-in replacement for the Digital Research command processor; nothing else in the system changes.

Initially I did this manually using the command

```
GET 100 :DU35.COM
```

to load DU35.COM into memory. Then I used a second GET command to load an assembled version of the dummy ENV and TCAP to the proper address:

```
GET 2D50 :CPMENV.COM
```

To make sure that the ENV address in this ENV module was self-consistent, I poked the correct value in at ENV+1BH:

```
POKE 2D6B 50 2D
```

Next, I had to poke the ENV address into the pointer at 109H. The command was:

```
POKE 109 50 2D
```

Now the memory image had the correct code, and I just

BYO Assembler, from page 20
is the "while" loop in which the condition is tested at the beginning of the loop, rather than at the end.

In Forth the accepted syntax for this structure is

```
BEGIN, evaluate cc WHILE, loop code REPEAT,
```

In practice, any code—not just condition evaluations—may be inserted between BEGIN, and WHILE,.

What needs to be assembled is this: WHILE, will assemble a conditional jump, on the inverse of cc, to the code following the REPEAT,. (If the condition code cc is satisfied, we should "fall through" WHILE, to execute the loop code.) REPEAT, will assemble an unconditional jump back to BEGIN. Or, in terms of existing constructs:

```
BEGIN,(1) ... cc IF,(2) ... AGAIN,(1) THEN,(2)
```

Once again, this can be implemented with existing words, by means of a stack manipulation inside WHILE, to re-arrange what jumps are patched by whom:

```
: WHILE, ( a cc - a a)   IF, SWAP ;
: REPEAT, ( a a - )   AGAIN,  THEN, ;
```

Again, nesting is freely permitted.

The Forth Definition Header

In most applications, machine code created by a Forth assembler will be put in a CODE word in the Forth dictionary. This requires giving it an identifying text "name," and linking it into the dictionary list.

The Forth word CREATE performs these functions for the programmer. CREATE will parse a word from the input stream, build a new entry in the dictionary with that name, and adjust the dictionary pointer to the start of the "definition field" for this word.

Standard Forth uses the word CODE to distinguish the start of an assembler definition in the Forth dictionary. In addition to performing CREATE, the word CODE may set the assembler environment (vocabulary), and may reset variables (such as MODE) in the assembler. Some Forths may also require a "code address" field; this is set by CREATE in some systems, while others expect CODE to do this.

Special Cases

1. *Resident vs. cross-compilation:* Up to now, it has been assumed that the machine code is to be assembled into the dictionary of the machine running the assembler.

For cross-assembly and cross-compilation, code is usually assembled for the "target" machine into a different area of memory. This area may or may not have its own dictionary structure, but it is separate from the "host" machine's dictionary.

The most common and straightforward solution is to provide the host machine with a set of Forth operators to access the "target" memory space. These are made deliberately analogous to the normal Forth memory and dictionary operators, and are usually distinguished by the prefix "T". The basic set of operators required is:

TDP	target dictionary pointer DP
THERE	analogous to HERE, returns TDP
TC,	target byte append C,
TC@	target byte fetch C@
TC!	target byte store C!
T@	target word fetch @
T!	target word store !

Sometimes, instead of using the "T" prefix, these words will be given identical names but in a different Forth vocabulary. (The vocabulary structure in Forth allows unambiguous use of the same word name in multiple contexts.) The 6809 assembler in Part 2 assumes this.

2. *Compiling to disk:* Assembler output can be directed to disk, rather than to memory. This, too, can be handled by defining a new set of dictionary, fetch, and store operators. They can be distinguished with a different prefix (such as "T" again), or put in a distinct vocabulary.

Note that the "patching" manipulations used in the single-pass control structures require a randomly-accessible output medium. This is not a problem with disk, although heavy use of control structures may result in some inefficient disk access.

3. *Compiler Security:* Some Forth implementations include a feature known as "compiler security," which attempts to catch mismatches of control structures. For example, the structure

```
IF, ... cc UNTIL,
```

would leave the stack balanced (UNTIL, consumes the address left by IF,), but would result in nonsense code.

The usual method for checking the match of control structures is to require the "leading" control word to leave a code value on the stack, and the "trailing" word to check the stack for the correct value. For example:

IF,	leaves a 1;
THEN,	checks for a 1;
ELSE,	checks for a 1 and leaves a 1;
BEGIN,	leaves a 2;
UNTIL,	checks for a 2;
AGAIN,	checks for a 2;
WHILE,	checks for a 2 and leaves a 3;
REPEAT,	checks for a 3.

had to save the file:

```
:SAVE 100-2E50 DU.COM
```

I presented the procedure above to illustrate how handy the Z-System GET, POKE, and transient SAVE.COM programs can be when doing this kind of work. (The peek command, P, also came in handy to let me see what I was doing.) To make the process easier for other people to carry out, I then developed a patch program called DU-CPM.Z80. Excerpts are shown in Listing 3, where the installation procedure is described.

Conclusion

I hope this column will inspire Z program authors to take a careful look at their programs to see how they can be made better behaved and more compatible with standard CP/M. I hope it will also inspire CP/M programs to think about the advantages that Z-System offers to many Z80 computer users and to devote the effort required to allow their CP/M programs to run effectively under Z-System as well. We will all benefit by bringing the CP/M and Z-System communities closer together.●

This will detect most mismatches. Additional checks may be included for the stack imbalance caused by "unmatched" control words. (The 6809 assembler uses both of these error checks.)

The cost of compiler security is the increased complexity of the stack manipulations in such words as ELSE, and WHILE,. Also, the programmer may wish to alter the order in which control structures are resolved, by manually re-arranging the stack; compiler security makes this more difficult.

4. *Labels:* Even in the era of structured programming, some programmers will insist on labels in their assembler code.

The principal problem with named labels in a Forth assembler definition is that the labels themselves are Forth words. They are compiled into the dictionary—usually at an inconvenient point, such as inside the machine code. For example:

```
CODE TEST ... machine code ...
     HERE CONSTANT LABEL1
     ... machine code ...
     LABEL1 NZ JP,
```

will cause the dictionary header for LABEL1—text, links, and all—to be inserted in the middle of CODE. Several solutions have been proposed:

a) define labels only "outside" machine code. Occasionally useful, but very restricted.

b) use some predefined storage locations (variables) to provide "temporary," or local, labels.

c) use a separate dictionary space for the labels, e.g., as provided by the TRANSIENT scheme [3].

d) use a separate dictionary space for the machine code. This is common practice for meta-compilation; most Forth meta-compilers support labels with little difficulty.

5. *Table Driven Assemblers:* Most Forth assemblers can handle the profusion of addressing modes and instruction opcodes by CASE statements and other flow-of-control constructs. These may be referred to as "procedural" assemblers.

Some processors, notably the Motorola 68000, have instruction and addressing sets so complex as to render the decision trees immense. In such cases, a more "table-driven" approach may save substantial memory and processor time.

(I avoid such processors. Table driven assemblers are much more complex to write.)

6. *Prefix Assemblers:* Sometimes a prefix assembler is unavoidable. (One example: I recently translated many K of Super8 assembler code from the Zilog assembler to a Forth assembler.) There is a programming "trick" which simulates a prefix assembler, while using the assembler techniques described in this article.

Basically, this trick is to "postpone" execution of the opcode word, until after the operands have been evaluated. How can the assembler determine when the operands are finished? Easy: when the next opcode word is encountered.

So, every opcode word is modified to a) save its own execution address somewhere, and b) execute the "saved" action of the previous opcode word. For example:

```
... JP operand ADD operands ...
```

JP stores its execution address (and the address of its "instance" parameters) in a variable somewhere. Then, the operands are evaluated. ADD will fetch the information saved by JP, and execute the run-time action of JP. The JP action will pick up whatever the operands left on the stack. When the JP action returns, ADD will save its own execution address and instance parameters, and the process continues. (Of course, JP would have executed its previous opcode.)

This is confusing. Special care must be taken for the first and last opcodes in the assembler code. If mode variables are used, the problem of properly saving and restoring them becomes nightmarish. I leave this subject as an exercise for the advanced student...or for an article of its own.

Conclusion

I've touched upon the common techniques used in Forth assemblers. Since I believe the second-best way to learn is by example, in part 2 I will present the full code for the 6809 assembler. Studying a working assembler may give you hints on writing an assembler of your own.

The *best* way to learn is by doing!●

References

1. Curley, Charles, *Advancing Forth.* Unpublished manuscript (1985).

2. Wasson, Philip, "Transient Definitions," *Forth Dimensions* III/6 (Mar-Apr 1982), p.171.

ADDITIONAL SOURCES

1. Cassady, John J., "8080 Assembler," *Forth Dimensions* III/6 (Mar-Apr 1982), pp. 180-181. Noteworthy in that the entire assembler fits in less than 48 lines of code.

2. Ragsdale, William F., "A FORTH Assembler for the 6502," *Dr. Dobb's Journal* #59 (September 1981), pp. 12-24. A simple illustration of addressing modes.

3. Duncan, Ray, "FORTH 8086 Assembler," *Dr. Dobb's Journal* #64 (February 1982), pp. 14-18 and 33-46.

4. Perry, Michael A., "A 68000 Forth Assembler," *Dr. Dobb's Journal* #83 (September 1983), pp. 28-42.

5. Assemblers for the 8080, 8051, 6502, 68HC11, 8086, 80386, 68000, SC32, and Transputer can be downloaded from the Forth Interest Group (FORTH) conference on GEnie.

Relay International Message Exchange Forth Conference

Edited by Gary Smith

RIME (Relay International Message Exchange) is a PC-board based network similar to FIDO. The original list *Forth* was produced by Bonnie Anthony, sysop of Running Board A—which is RIME central. Hers was only a node list. The Forth conference originates on Jim Wenzel's Grapevine BBS in Little Rock, AR. Jim was able to identify all but a few nodes and affix the phone number and system operator associated with each. It's a rather impressive list.

Messages carried on RIME Forth conference are essentially identical to those carried on the GEnie Forth RoundTable and on UseNet comp.lang.forth news feeds. Our apologies for any systems not mentioned.

Abstract BBS	ABSTRACT	Mark Froese	(718) 351-7633
Alpine BBS	ALPINE	Jason Hills	(503) 581-0923
Aquila BBS	AQUILA	Steve Williams	(708) 820-8344
Baudline II	BAUDLINE	Drew Bartorillo	(301) 694-7108
Castle Rock BBS	CASROCK	Rocco Fili	(402) 572-8247
The Computer Forum BBS	CFORUM	Jim Rhodes	(804) 471-3360
Cloud Nine BBS	CLDNINE	Ed Lucas	(713) 859-8195
The San Diego CLIP*BOARD	CLIPBORD	Ted Blue	(619) 427-4664
Club PC BBS	CLUBPC	Jim Kreyling	(804) 357-0357
Computronics Comm Link	COMPTRON	Ken Hunt	(813) 526-1265
The Consultant's Forum	CONFORUM	Bill Raines	(513) 424-1861
Canada Remote Systems	CRS	Jud Newell	(416) 629-0136
DFW Programmer's Exchange	DFWPGMR	Ric Naff	(214) 398-0013
Digital Schoolhouse	DIGISCHL	Richard Munro	(416) 299-7306
The Enchanted Forest BBS	ENCHANT	David Rockey	(904) 377-2001
The Godfather	GODFTHR	Kathi Webster	(813) 289-3314
The Grapevine BBS	GRAPEVNE	James Wenzel	(501) 753-8121
Gwinnett Hospitality	GWINNETT	Gene Hysner	(404) 962-6820
Hallucination BBS	HALCNTN	Michael Malak	(703) 425-5824
the Haven of Rest	HAVEN	Byran Pike	(612) 474-0724
The Holistic BBS	HOLISTIC	Mike McCarthy	(213) 531-3890
Data Central BBS	ICC	Doug Scott	(317) 543-2000
The Icebox BBS	ICEBOX	Darren Klein	(718) 793-8548
Life Scan Communications	LIFESCAN	Kelly Tompkins	(213) 865-3988
HHDC BBS	MEDINFO	Peter Booras	(904) 221-9425
The Mog-Ur's EMS	MOGUR	Thomas Tcimpidis	(818) 366-1238
Moondog	MOONDOG	Don Barba	(718) 692-2498
M.O.R.E.	MORE	JThomas Howell	(401) 849-1874
The Musical Chair	MUSICAL	Jeff Woods	(416) 438-3009
Network East	NETEAST	Howard Hartman	(301) 942-5616
O.L.E.F.1	OLEF	Peter Grain	44-81-882-9808
Programmer's Palace	PALACE	Matthew Briggs	(703) 866-4452
PDS-SIG BBS	PDSSIGI	Bob Allen	(408) 270-4085
The Pegasus BBS	PEGASUS	Raymond Clements	(502) 684-9855
ST. Pete Programmer's Exchange	PETEXCH	Bill Blomgren	(813) 527-5666
PGHSouth PCBoard System	PGHS	Dan Deady	(412) 563-5416
ProPC BBS	PROPC	Robert Malakoff	(412) 321-6645
The Punkin Duster BBS	PUNKIN	David Ludwig	(714) 522-3980
PC Rockland BBS	ROCKLAND	Charlie Innusa	(914) 353-2157
The Round Table BBS	ROUND	Daniel McCoy	(215) 678-0818
The Running Board	RUNNINGA	Bonnie Anthony	(301) 229-5623
The Cave	THECAVE	Roger Lee	(408) 259-8078
The Pub BBS	THEPUB	Jim Fennell	(914) 686-8091
The TREE BBS	THETREE	Frank Fowler	(904) 732-0866
Technet At TJHSST	TJHSST	Kelly Deyoe	(703) 941-3572
The Right Place (tm)	TRP	Roger Sligar	(404) 476-2607
The Virginia Connection	VIRGIN	Tony McClenny	(703) 648-1841
Washington PC-Board	WASHPCB	Mike Keelon	(412) 225-9782
The Windows Plus BBS	WINDOWS	Russell Jackson	(501) 968-8431

YASBEC, from page 7

What's next

Designing a project like YASBEC is like writing a program. There are as many ways to get the job done as there are designers. Looking back I always see things I wish were done different. Unfortunately you can not simply 'edit' hardware like software, once the choices are made you must live with them. Luckily after months of playing with this board I have found little I would have done differently. But where do we go from here? Well I currently have photo tools ready for a backplane and a memory expansion board. But why a backplane for a single board computer? Because the one with the most toys when he dies wins! Seriously, while the YASBEC makes a nice system all by itself, it does lend itself to all sorts of possibilities with a backplane. YASBEC can support two 512K x 8 static RAM chips, but these chips currently run around $250 each! Next year they may be affordable but for now a memory board would be nice with several 128K x 8 chips. The memory board also includes a non-volatile controller and battery for all the RAMs making a nice RAM disk. And how about video? Wayne and I currently have a prototype video board running. It has resolution and colours similar to VGA but with a RAMDAC to provide 256 x 24 bit mapped colour. It also has a 256 gray scale real-time frame grabber. The frame grabber still needs work but results so far are promising.

Backplane

Figure 6 shows the backplane pin out and driver chips. The pin out of the backplane was determined largely by the circuit board layout. While the Eurocard PCB, card cage and connector are standard the pin out is unique to YASBEC. U7 provides the bi-directional data bus while the other four 74ACT245 chips buffer the address and control lines. Now, how many hardware gurus noticed something strange? There are four bi-directional bus drivers hard wired uni-directional. Why the overkill? Well, I had lots of 245 chips in surface mount.

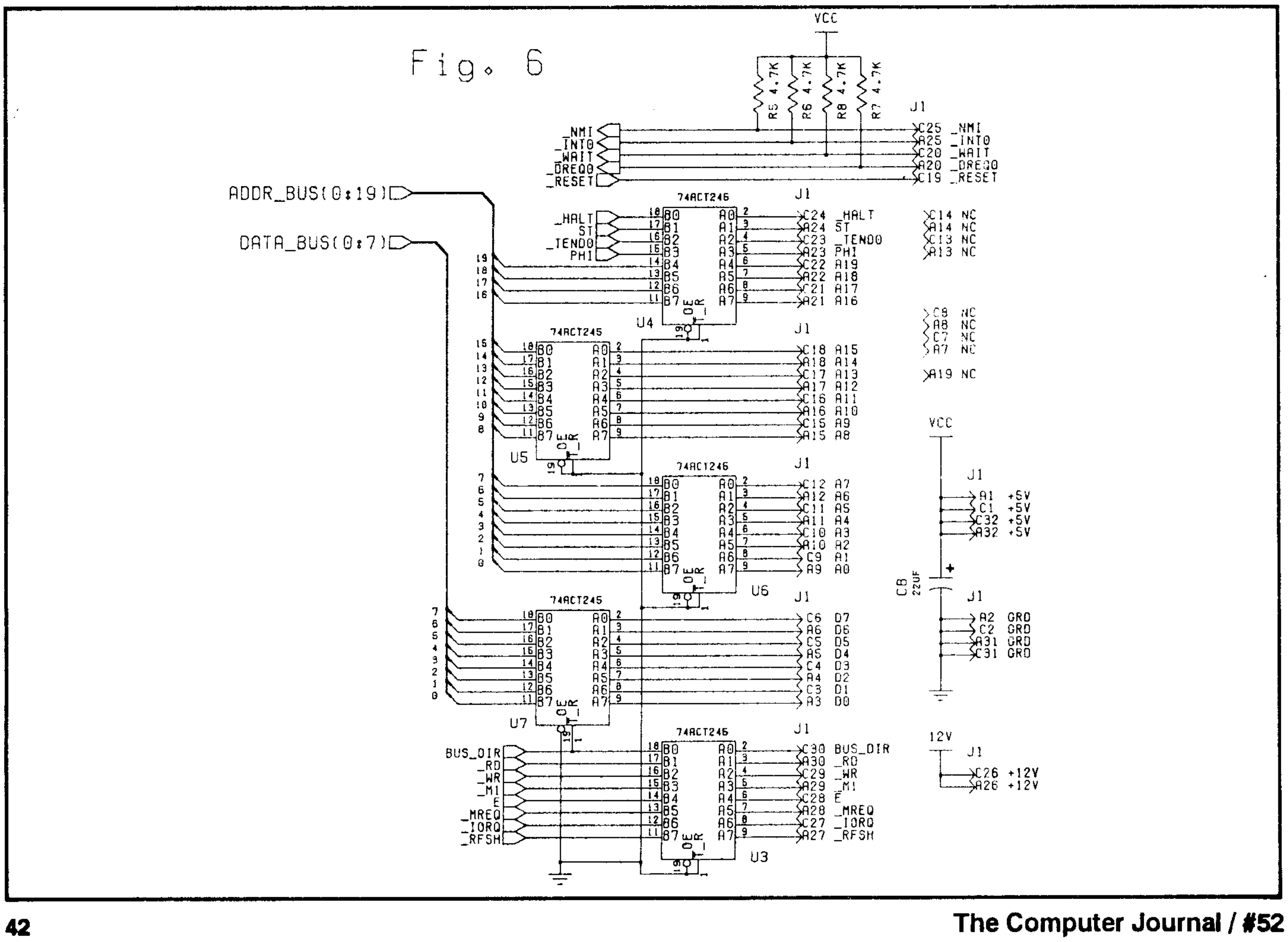
Software

Software for the YASBEC is still under development. Besides a standard 2.2 system and BIOS, a banked system is also currently under way. By the time this is in print we should have an answer for those people who do not want to port CP/M themselves. The best place to stay up to date is on GEnie. [Ed.: There are several topics on YASBEC in the CP/M SIG].

Where do I get one?

The YASBEC is currently available as a semi-populated kit for CDN$100 plus $10 shipping and handling for Canada and the USA. Payment by cheque or postal money order, sorry no credit cards. You can send your order to my address as given in the *biographical* paragraph to this article. Included in the price are the monitor ROM, address decoding PALs, and all surface mount components soldered to the board. Not included are the sockets, socketed parts, cables and any leaded parts (jumpers, connectors, et cetera). Along with the parts list I have included suggested sources, I'll be glad to help if you have trouble finding parts. Assembling this system should prove itself an interesting project for the true hacker. Just like the good old days, eh? Anyone interested in a fully assembled system in a case with power supply, console and software ready to boot should go buy a PC.●

Fig. 6



Z-Best Software

What Summer Doldrums?

By Bill Tishey

I've been extremely busy the past few months as Z-System Librarian recording a multitude of Z program releases. Therefore, instead of tips for updating the Z3HELP system, which I promised in the last column, I thought it more appropriate this time to list for you the recent updates and new releases. See *Listings 1* and 2 for the many "new" and "revised" programs. I'll comment briefly on some of these and describe a few in more detail. The listings, by-the-way, are taken from my ZFILEV10.LST. Note the new format for the 3-line description which assigns a category to the program and indicates whether a HELP (.HLP) and ZCNFG configuration (.CFG) file are available for it. I'd like to hear from readers about what they'd like to see in this column. Such listings could be a continuing feature. Let me know what you prefer.

Listing 1

```
New Releases:

Name          Vers   S ZSUS Siz Rec  CRC Library/Size Issued   Author
=====================================================================
DIALER.COM   1.00    4          2   10 72D3 DIALER      9 07/23/91 Bruce Morgen
  BBS    Allows your computer and Hayes-compatible modem to dial your voice
  HLP=N  telephone calls.  Helpful for scripting through banking and other
  CFG=N  touch-tone-based response systems.

NTS.COM       1.00    3 V2/10  5   27 E831 NTS10       9 06/01/91 Rob Friefeld
  DATE   Modified version of NT (Note Taker) which has a simple full screen
  HLP=Y  editor.  WordStar command set.  Appends short notes to files,
  CFG=Y  including the current time.

PUSHDIR.COM  1.00    4          1    8 2CA4 PUSHDIR     8 06/11/91 Bruce Morgen
  DIR    Saves the current default user code and drive to user registers 0 and
  HLP=N  1 respectively and restores this location as needed later.  Works
  CFG=N  under all variants of ZCPR3, Z-System, and BGii.

TXTALIAS.COM 1.00    0 V2/10  4   29 88D1 TXTALIAS  17 05/20/91 Bruce Morgen
  ALIAS  "Compiles" standalone aliases from standard ASCII source files.
  HLP=Y  Similar to ACREATE and BA24, but supports assembly-style comments and
  CFG=N  multiple linefeeds in the source file.  Uses latest, built-in ALIAS0.

ZDT.COM       0.90    0 V2/10  8   64 CE56 ZDT09       9 07/10/91 Joe Mortensen
  DBASE  Z-System Day Timer, a daily planning calendar derived from ZDB.
  HLP=N  Automatically reads the real-time clock and displays the current
  CFG=N  day's schedule.  Requires ZCPR30+ and extended TCAP.

ZFIND.COM     1.30    0 V2/10  4   29 8F94 ZFIND13    34 08/01/91 Terry Hazen
  FILE   ZCPR3 string search utility which very quickly finds ASCII strings in
  HLP=Y  text files.  Found string can be displayed in either line or delimited
  CFG=Y  block.  Output can be written or appended to a file.
```

Hal Bower: Hal has released Version 4.4 of the Libraries which consolidates changes to DSLIB, SYSLIB, VLIB, and Z3LIB since version 4.3 (LIBS44A.LBR in Microsoft format and LIBS44AS.LBR in SLR format). The major improvements have been to Z3LIB's NDR routines. Some additional work on DSLIB still remains.

Gene Pizzetta: Gene has made some minor bug fixes to CONCAT (a P2DOS bug) and DATSTP (display of European date with the 'D' option). Also, his ROMAN utility now converts either way (between roman numerals and decimals) and will also run under vanilla CP/M. Gene is now working on an update of ZSLIB (version 3.0 should be available as you read this). Incompatibilities with DSLIB are being corrected, command line parsing is being improved, and many new routines are being added for date and time output.

Bruce Morgen: Bruce has upgraded a number of utilities to Type-4 status (TRIM, W), and continues to improve others to run more effectively when linked as Type-4s (SETPATH, VREN). He's added LZH support to the LUSH library shell and has done some code-crunching on CPA, LGET and PATH. He's also implemented in the Z34 transient POKE utility word-wide operations when a hex number of 3 or more digits is used. This is useful with ALIAS17 and SALIAS15 and their new

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.

pointer capabilities, which return word-wide values. POKE can now pass such values to system registers for use for flow control in subsequent aliases or with RESOLVE, etc.

Bruce has also introduced DIALER, a nifty little program to automate voice telephone calls to touch-tone-based response systems. DIALER-based Z-System scripts can relieve one of the hassle of remembering long banking access numbers, etc. It is hardware-independent and assumes only that data can be sent to the modem via the PUN: device and that the modem will allow blind dialing (no dial tone or busy signal detection) and will not wait for a carrier before going back on-hook.

Joe Mortensen: Joe continues to tweak his Z-System name and address database, ZDB. Version 1.4 is considerably faster in finding records and fixes some problems in screen display.

Joe has also introduced ZDT, the Z-System Day Timer. A derivative of ZDB and still undergoing evaluation and testing, ZDT is a daily planning calendar which automatically reads the real-time clock and displays the current month's calendar and schedule for the current day. Word has it that Joe is now working on ZBIB, a "bibliographic database" patterned after ZDB.

Terry Hazen: Terry has undoubtedly kept busy this summer with updates to ACOPY, DSTATS, NZBLITZ, REMIND, SCAN, and a new release - ZFIND (see below).

At the urging of Howard Schwartz, Terry has expanded the status message of ACOPY's 'Update' option to indicate whether an older file as been replaced or not when a file is over-written. A file with no datestamp is automatically 'Dated', while one with an older stamp is 'Updated'. An update failure now either produces the message "No Update (Source Older)" or "No Update (Same Date)". DSTATS, a derivative of DSKMAP and UMAP, now displays the following additional disk statistics: the maximum number of disk directories, (in each user-area display line) the amount of space used, and (in the "Free:" summary line) the amount of disk free space. NZBLITZ, which saves and loads NZCOM system images, is now ZCNFG-urable, allowing a choice of image saves 1) up to CBIOS, 2) up to a specified address, or 3) up to the top of memory. The NZCOM.CCP file produced by the NZBLITZ loader can also be flagged as

Listing 2

```
Revised Programs:

Name          Vers   S ZSUS Siz Rec  CRC Library/Size Issued   Author
======================================================================
ACOPY.COM     3.30   3         6  46 EB5F ACOPY33   26 07/21/91 Terry Hazen
  FILE   Attribute Copy program.  General file copy tool with source and
  HLP=Y  destination disk directory caching.  ACOPY.PAT has patch locations.
  CFG=N  Derived from PPIP vs 1.2.

CLED          1.50   4        na  na na   CLED15    38 07/23/91 Rob Friefeld
  RCP    ZCPR34 resident command line editor.
  HLP=Y
  CFG=N

CONCAT.COM    1.40   0         7  52 B373 CONCAT14  43 05/29/91 Gene Pizzetta
  WP     Concatenates two or more source files into a destination file,
  HLP=Y  similar to PIP, or appends them to an existing file.  Accepts both
  CFG=Y  DIR and DU specs.  Checks for adequate disk space.  For ZCPR3 only.

CPA.COM       1.30   0         4  32 49B4 CPA13     20 07/05/91 Bruce Morgen
  FILE   ComPare Ascii utility to analyze two files, on a line by line basis,
  HLP=Y  and report any differences.  Vs 1.0 (01/87) by Malcolm Kemp.
  CFG=N

DATSTP.COM    1.50   0         5  37 A66B DATSTP15  70 05/27/91 Gene Pizzetta
  DATE   Displays or changes the create and modify date stamps on any file
  HLP=Y  from the command line.  Universal version (ZSDOS/ZDDOS/ZRDOS with
  CFG=Y  DateStamper, under Z3PLUS will display but not change datestamps).

DSLIB.REL     4.40   4 V2/11   6  44 3905 LIBS44A   65 06/30/91 Hal Bower
  PROG1  Routines to facilitate addition of file time- and datestamping and
  HLP=Y  real-time clock features.  Microsoft REL format.
  CFG=N

DSTATS.COM    1.10   0         2  16 DF42 DSTATS11  18 07/23/91 Terry Hazen
  DISK   ZCPR3 disk/user statistics utility.  Displays disk block size, disk
  HLP=Y  capacity, allocated and free space, list of active user areas, etc.
  CFG=Y  Combines DSKMAP and many functions of UMAP.

DU.COM        3.50   0        12  91 496C DU35      62 07/21/91 Jay Sage
  DISK   ZCPR3 Disk Utility.  Similar to vs .4, but can be patched to operate
  HLP=N  under standard CP/M as well as Z-System.  See DU.COM vs 3.14.
  CFG=N

HELPC.COM     1.30   0         5  36 98B9 HELPC13   23 06/20/91 Howard Goldstein
  HELP   Replacement for the standard Z-System HELP utility.  Handles crunched
  HLP=Y  as well as normal, uncompressed help files.  Print options are
  CFG=Y  disabled when wheel byte is turned off.  Configurable with ZCNFG.

JETCP.COM     1.20   0         4  26 E78F JETCP12   17 06/08/91 Carson Wilson
  FILE   A fastest-possible speed file copy program for Z3PLUS only.  Multi-
  HLP=Y  sector I/O and advanced BDOS error-handling.  Copy buffers are as
  CFG=N  large as memory will allow.  Preserves CP/M Plus datestamps.

LGET.COM      1.30   0         4  29 3B20 LGET13     4 07/20/91 Bruce Morgen
  LBR    Extracts specified files from an indicated LBR.  Vs 1.0 by R. Conn.
  HLP=Y
  CFG=N

LUSH.COM      1.20   4         4  26 1150 LUSH12    24 07/26/91 Bruce Morgen
  SHELL  Library Utility SHell which takes advantage of the extended services
  HLP=Y  of ZCPR33+.
  CFG=N

NAME.COM      1.20   4 V2/10   2  11 E793 NAME12     9 06/30/91 Bruce Morgen
  NDR    Renames and deletes directory names on-the-fly.  Inspired by Jay
  HLP=Y  Sage's ARUNZ script for the same function.
  CFG=N

NZBLITZ.COM  1.40   4 V2/10   2  15 A97B NZBLTZ14  22 06/23/91 Terry Hazen
  SYS    NZCOM utility which saves and loads system images.  Allows fast cold
  HLP=Y  load of a system - full up with desired drivers, path, options, etc.
  CFG=Y  Configurable with ZCNFG.  Vs. 1.0 by Cam Cotrill.

PATH.COM      3.20   4         2  12 08CE PATH32     9 06/07/91 Bruce Morgen
  SYS    Allows the user to display the current path or set a new path.
  HLP=Y  Command line quiet option.  Z80 required.  Vs 3.0 (4/12/84) by R.
  CFG=N  Conn.
```

```
POKE.COM      1.10   4 V2/10  2   9 A84B POKE11      7 07/12/91 Bruce Morgen
  SYS    Transient replacement for the RCP-based POKE command.  Type 3 at
  HLP=Y  8000h.
  CFG=N

REMIND.COM    1.40   0        5  40 3FB7 REMIND14   36 07/21/91 Terry Hazen
  DATE   Appointment reminder utility for ZCPR3 and ZSDOS with clock.  Will
  HLP=Y  display and optionally print a sorted and paged list of dated appt
  CFG=Y  reminder lines with optional time entries from a text datafile.

ROMAN.COM     1.00   4 V2/10  2  16 4000 ROMAN10    14 06/01/91 Gene Pizzetta
  WP     Z'ified version of ROMAN.COM (10/78 by M. Pedder) which converts
  HLP=Y  between decimal numbers and Roman numerals.  Works from command line
  CFG=Y  or in interactive mode.

SCAN.COM      2.40   0        6  47 46C0 SCAN24    100 06/12/91 Terry Hazen
  FILE   Bi-directional, video-oriented text file display utility that uses
  HLP=Y  the basic WordStar command set to control viewing.
  CFG=Y

SCOPY.COM     0.60   4       15 114 B6AB SCOPY06    30 06/01/91 Rob Friefeld
  FILE   Screen-oriented file-copy utility.  Displays source and destination
  HLP=Y  directories in vertical windows.  ZF-like commands.  Supports file
  CFG=Y  selection and copy by datestamp.  Extended TCAP required.

SETPATH.COM 1.10   4        4  27 D71E SETPTH11   20 06/07/91 Bruce Morgen
  SYS    Enhanced derivative of PATH.COM which allows adding/deleting elements
  HLP=Y  from either end of the path.  Vs 1.0 (3/8/87) by R. I. Demrow; derived
  CFG=N  from PATH vs 3.0 (04/84) and PATH 1.0 (01/83) by R. Conn.

SYSLIB.REL    4.40   4 V2/11 22 173 7F7B LIBS44A    65 06/15/91 Hal Bower
  PROG1  Upgrade of SYSLIB 4.0 for Z80 compatible computers.  Microsoft REL
  HLP=Y  format.
  CFG=N

TRIM.COM      1.10   4        2  10 875F TRIM11      8 07/09/91 Bruce Morgen
  PROG2  Truncates .COM files at a requested address.  Used to delete
  HLP=Y  unneeded DSEG from the output of linkers like LINK, L80, ZLINK.
  CFG=N  Type 3 at 8000h.

UNZIP.COM     1.50   C       15 119 8D13 UNZIP15    23 06/01/91 Howard Goldstein
  LBR    Extracts all members of a specified .ZIP file matching <afn>.  If
  HLP=N  <afn> is not present, a directory of the .ZIP file is displayed.
  CFG=N

VLIB.REL      4.40   4 V2/11  6  44 DD58 LIBS44A    65 06/16/91 Hal Bower
  PROG1  Library routines for basic CRT screen manipulation as well as extended
  HLP=Y  graphics routines.  Microsoft REL format.
  CFG=N

VREN.COM      1.10   4 V2/10  4  27 3A4A VREN11     31 06/05/91 Bruce Morgen
  FILE   Visual File RENamer.  Provides interactive renaming of files with
  HLP=Y  format checking.  Vs. 1.0 (8/8/88) by Bruce Morgen.
  CFG=N

W.COM         2.40   4        4  32 A6F2 W24        30 07/03/91 Bruce Morgen
  FILE   Wildcard Shell Processor that enables wildcard processing for
  HLP=Y  programs that do not usually accept wildcard parameters.
  CFG=N

XOX.COM       1.0k   4       12  93 C359 XOX10K     27 06/01/91 Rob Friefeld
  FILE   Text file viewer with additional functions for listing blocks, writing
  HLP=Y  blocks to disk, & merging files.  Does not uncompress files or access
  CFG=N  libraries.  Follow-on to VIEW.COM.  Requires extended TCAP.

Z3LIB.REL     4.4a   4 V2/11 11  88 A768 LIBS44A    65 07/14/91 Hal Bower
  PROG1  ZCPR3-specific library routines.  Microsoft REL format.
  HLP=Y
  CNG=N

ZDB.COM       1.40   0        8  64 DB21 ZDB14      31 06/28/91 Joe Mortensen
  DBASE  Small, fast name and address file manager with built-in label and
  HLP=Y  envelope addressing features.  Requires extended NZTCAP.
  CFG=Y

ZFILER.COM    1.0p   4       15 116 51B7 ZF10P     105 07/27/91 Rob Friefeld
  FILE   Enhanced version of VFILER designed to take advantage of ZCPR33+
  HLP=Y  facilities.  5-col, reverse video, DateStamper version.
  CFG=N
```

a system file and/or archived file. Any pending multiple command line in the old system is now saved and appended to the loader MCL, a feature which should allow for some interesting use of loader files in aliases.

Major enhancements to REMIND are the inclusion of a calendar display at startup, along with the option ("/1-12") to display a calendar and upcoming reminders for a specified month only. Screen paging is also now an option (allowing it to be turned off when sending reminders to a printer).

Terry has also added speed, smoothness and an improved screen display to SCAN, his bi-directional text-file viewer. Use of a modified Boyer-Moore search algorithm also now results in faster string finds. Rob Friefeld: Besides a significant update to ZFILER (see below), Rob brings us NTS, a modified version of Note Taker, as well as updates to CLED (see below) and his screen-oriented utilities SCOPY and XOX. Rob is also working on enhancements to LSH, his now-famous command-line history shell.

Improvements to SCOPY include a new command to RELOG the current source/ destination/mask, some fixes to the Zip command, and a new Group Zip command to copy all tagged files to an alternate directory. "Z" and "GZ" now act more like ZFILER's "C" and "GC" commands.

ZFIND.COM vs 1.3 (ZSUS Vol 2 #10)

Terry Hazen's new ZFIND utility is a string-find tool styled after Irv Hoff's FIND.COM, only *much* faster and more powerful.

ZFIND will find one or more ASCII strings in a group of wildcarded ASCII text files. Searches are performed in a 16k text buffer, using a modified version of the Boyer-Moore string-search algorithm. When a matching string is located in the buffer, the line containing the string is displayed, in either the line or the delimited block in which it was found. The entire buffer is first searched for string1 and the finds displayed, then the buffer is searched for string2 and those finds displayed, etc. When the buffer has been searched for all strings, the next buffer-full is read in and the process repeated. The same process is repeated for each file that matches the specified ambiguous filename. Since the search process ignores lines except to display the found string, line number output is not provided.

```
ZP.COM       1.40   4       8  64 5BD8 ZP14       95 07/17/91 Terry Hazen
  FILE   ZCPR33+/Z3PLUS screen-oriented file/disk/memory record patcher using
  HLP=Y  the ZPATCH command set.  Requires a VLIB4D+ Z3TCAP.  One-record
  CFG=Y  cache can be exchanged with file/disk/memory records.

ZS.COM       1.10   0      14 105 FF2A ZS11       47 06/28/91 Pete Pardoe
  LBR    Z'ified version of Dave Rand's NSWP207 with NDR support and ability to
  HLP=N  act upon the current file with any Z-System command up to 127 chars.
  CFG=N  Requires ZCPR33+ with extended TCAP.
```

See *Figure 1* for ZFIND's syntax.

Certain characters may be used during string entry to enter special characters into the search string: a question mark matches any character in that position; an underline character is changed to a TAB; and a backslash is changed to a line feed, allowing searches for strings starting at the beginning of a line. Output can also be written or appended to a file. Searches are normally performed ignoring case; however, if you don't specify a string on the command line, one is requested, allowing you to enter the string in the exact case desired. Terry provides several sample applications for ZFIND in his included help file (ZFIND13.HLP). Single-line index files are simplest to search. The alias "LOOKUP zfind dir:mast.cat $*", for example, will perform fast filename lookups in MAST.CAT files. Block-display applications, however, are also easy. The 'B' option, for example, allows lookup of keywords in *.FOR files, displaying the entire FOR messages for a specified filename. The same option can be used to search ZFILEVxx.LST, RCPMxxxx.LST and similarly delimited lists. ZFIND's faster search speed and ability to output/append to a file would appear to make it more powerful than XFOR for such applications.

Figure 1

```
B0:WORK>ZFIND //

ZFIND String Find Utility vers 1.3
Syntax:
   ZFIND [dir:]afn [string(s)] [>outfile] [/options]
   If no string is included, one will be requested,
     permitting a search for lower-case characters.
   Special string characters:
        "|" separates multiple search strings
        "?" matches any character
        "_" matches a tab character
        "\" matches the beginning of a line
Options:
   U - Search files on all user areas
   A - Display all files searched
   C - Display found string line in context (3 lines)
   B - Display found string line in a delimited (LF,"-") block
   D - Don't Page screen display
   >[dir:]outfile  - Output to file
   >>[dir:]outfile - Append to file
```

Since ZFIND is ZCNFG-urable, default configurations can be created for a wide variety of applications. To create a 'card' file of topic or article paragraphs, for example, where each paragraph contains keywords and topic summaries, descriptions, etc., simply configure ZFIND to use 0Dh (CR) as a delimiter. ZFIND will then display the matched strings in single-spaced paragraphs. To create a version of ZFIND which always searches all user areas on the specified or default drive, simply set the 'U' option test to always true.

ZFILER vs 1.0P

Rob Friefeld's update of ZFILER, the ZCPR3 file-maintenance shell, has been much anticipated. The last update (vs 1.0o) by Carson Wilson was nearly two years ago, so there was much on Rob's list for improvement. Besides fixes to some pesty filecopy bugs, the following changes and additions are worth noting:

1. ZF10P.LBR is a hefty file (105k), and includes the usual separate versions for display of 4 or 5 columns of files, dim or reverse video, and support for DateStamper. It also, however, contains 1) a complete version history since its predecessor, VFILER 4.2, 2) the text of Jay Sage's *TCJ* articles (issues 36 and 37) detailing the functions of ZFILER, and 3) an addendum updating Jay's material. The latter contains notes on Group Macro scripting which many have been seeking since version 1.0m. Such attention to documentation was particularly thoughtful, but then, this has been a trademark of all of Rob's work!

2. ZFILER is now ZCNFG-urable. An included ZCNFG (.CFG) file handles complete configuration except for macro string installation. Three screens allow setting of general defaults as well as specific "macro-related" options. "General" options include a toggle for uppercase display of filenames and another for "clear screen on exit". (The latter should be good news to those who've ever run a macro command from ZFILER.CMD, then, as ZFILER reloaded with the "Strike Any Key" prompt, watched it clear the screen of any message from the program that had just run before you could read it!) "Macro-related" options allow setting of the search path or a fixed DU: for ZFILER.CMD and ZEX batch files. Another option allows erasing of a .ZEX file after a shell run. (Of course, you'd only want to set this once you've debugged your macro command file!)

3. ZFILER's Group Tag/Untag commands now operate from the file pointer to the end of the file list, rather than on the entire list. Thus, if you go to mid-screen and press "GT", only files from the cursor position to the end of the file list will be tagged. This is a restoration of how ZFILER used to work.

4. The Group Macro now puts the tagged file count into a configurable user register before running. This allows the macro to downcount and take some additional action after it has run on each tagged file. As a sample application, Rob includes the following macro, which creates libraries of crunched files (thanks to Lindsay Haisley for pushing for this feature!):

```
1 ! $d$u:;$!crunch $f M1:;reg m5;if reg 5= 0;$"Library DU ":;
    lput $"Library Name: " M1:*.* +1;era M1:*.?Z?;fi;$h:
```

Figure 2

```
Syntax:  CLED
            CLED [/]
             If "/", run once only (e.g. from a shell such as ZFILER)

  The default control key set includes these important commands:

     CR - execute command line
   ESC Q - pop the shell
   ESC S - toggle recording on/off  (prompt shows >/>> to indicate state)
      ^W - recall command lines from history stack
      ^E - recall history in reverse direction
```

Typing "G<ESC>1" will crunch all tagged files to a fixed directory M1: (on a RAM disk), make a library out of them (leaving room for one more file) with input for library name and location, then erase all the crunched files. [Note that REG must be a Type3 or Type4 program (or RCP) for the ZEX GO "$!" to run CRUNCH repeatedly. Also remember that registers only hold a 255 count! For consistency, single macros put a "1" count in the user register. ZFILER will also allow use of the full register range (0-31), although one should be aware that 10-15 are reserved and 16-17 are used by other programs.] The only complaint remaining about ZFILER which I've heard of is its failure to filter output generated by the view function. Certain escape sequences when not trapped can wipe out a terminal (even reset the configuration stored permanently in RAM). Rob may address this problem in a future update.

CLED vs 1.5

Rob has also updated his memory-resident command line editor (CLED), originally distributed with Z34RCP11.LBR. Installed as part of an RCP, CLED adds the features of cursor movement, insert/delete, and record/recall of command lines to your Z34 CCP. It is a ZCPR "shell" (using up one shell stack entry), and thus will reinvoke itself automatically following any ZCPR command. If a ZS/ZDDOS, DateStamper, or Z3PLUS clock is implemented, the command prompt will show the system time. See *Figure* 2 for CLED's command syntax.

CLED15.LBR includes two companion programs. An installation program, CLEDINST, allows you to set up CLED to your preference. The RCP can be installed directly in memory for testing, and then saved to a file with an image-saver such as SNAP or NZBLITZ. The segment image can also be saved with command lines already loaded.

CLEDSAVE, a history save/load tool, writes the contents of the history stack to a text file on disk. This file can be reloaded later (CLEDSAVE <file> L), or composed in advance with a text editor, then loaded. CLEDSAVE is useful in a startup alias to load frequently used command lines from an easily edited file.

See the *Z-Message Base* below for instructions on assembling an RCP to implement CLED. Note that the RCP must be assembled from scratch, loaded with JETLDR, *then* saved with NZBLITZ. You can't simply load the new RCP into an old NZRCP.ZRL file. If you don't have an assembler, there are many users who would be happy to create the CLED-RCP segment for you. Simply decide what you want in your RCP and how many records you have allotted and drop a note to the editor or myself with your system specifics.

Z Message Base

Z-Node #2, 07/26/91 CLED15.LBR

Question: I'm interested in implementing the improved CLED editor. What do I assemble the RCP with and how do I select the part to be assembled?

Answer: Here are the steps to implement CLED15 in a Z34 RCP. The SLR assembler or ZMAC will do.

1. You need all the source code distributed with Z34RCP11.LBR.

2. Edit Z34RCP.LIB

Find the configuration information "CLED Command" and replace it with Z34RCP.INC in the CLED15 library. Look through the section to set up the history size you want. (Kaypro users should put in the Kaypro EREOL screen code (18H?) to save lots of bytes.)

3. Assemble the RCP:

```
A0:>SLR Z34RCP11/6
```

The assembler will prompt you for which modules to include. (My 18 record RCP has room for CLED, ECHO, ERA, CLS, and some little custom ditties.) ZMAC Z34RCP11 also works, but the prompts are a bit scrambled.

Just type a Y or N if you want a module or not.

4. Load the .REL (rename to .ZRL if you prefer) file with JETLDR and it is running.

5. Run CLEDINST to install your command set in the RCP.

6. Save it with NZBLITZ or SNAP. (Rob Friefeld) ●

Reader, from page 2

ommend) speaks of the need for "value-added differentiation"—some difference that makes your product more valuable than the others. I think *TCJ* has this....but it took me over an hour (and two drafts) to be able to put even part of it in words.

Perhaps *TCJ*s problem is image. Peters also points out that successful enterprises need a vision: a short, clear, understandable statement of mission and purpose. I think that, until the last issue, *TCJ*s vision has been a bit murky; and this translates into prospective subscribers not being really sure what *TCJ* is for. (Issue 51 has started to change this, with the ad on the back and also with your *Editor's Desk.*) A clear vision is a message you can promote, an idea you can sell.

Some famous producer once said, "if you can't write your [story] idea on the back of my business card, you don't have a clear idea." It's a good target to shoot for.

B.R. Toronto, ON

You identified the TCJ reader quite well: embedded systems developers, serious hobbyists, experienced CP/M users and novice Forthers. This is a market I am comfortable with. However, you say, "low-budget hackers who 'cross-specialize' in both hardware and software, who like to tinker and know everything about how their computers work," (people like yourself) are a dying breed. Is this to say "the barn storming days are over?" Look at any wheat field in Kansas. There are still biplanes earning a living every day of the year. We may not represent the main stream, but we are and always will be a necessary part of the whole. As long as there are chips on boards and boards in systems, someone has to know what they do.

Luckily, TCJ doesn't need many such people to be a commercial success. BYTE has a circulation of 450,000. One percent of this would be a good number. Do you think in all the computer industry there are still 4,500 people like us?

"If you can't write your idea on the back on my business card, you don't have a clear idea." Try this:

"TCJ serves the entrepreneur in embedded controls and the serious computer hobbyist desiring to learn software and hardware

See Reader, page 50

TCJ The Computer Journal Market Place

THE STAUNCH 8/89'er

Support for Heathkit H-8 and H/Z-89 Computers:

Bimonthly Newsletter, Z-System & HDOS Systems, CP/M and HDOS Utility and Applications Software.

THE STAUNCH 8/89'er
c/o Kirk L. Thompson
P.O. Box 548
West Branch, IA 52358
Voice: 319-643-7136
(eves and weekends)

Advent Kaypro Upgrades

TurboROM. Allows flexible configuration of your entire system, read/write additional formats and more. $35

Hard drive conversion kit. Includes interface, controller, TurboROM, software and manual—Everything needed to install a hard drive except the cable and drive! $175 without clock, $200 with clock.

Personality Decoder Board. Run more than two drives, use quad density drives when used with TurboROM. $25

Limited Stock — Subject to prior sale

Call 916-483-0312 eves/weekends or write Chuck Stafford, 4000 Norris Avenue, Sacramento CA 95821

TCJ The Computer Journal Market Place Advertising for Small Business

Looking for a way to get your message across? Advertise in the Market Place!

First Insertion:	$50
Reinsertions:	$35

Rates include typesetting. Payment must accompany order. Foreign orders paid in US funds drawn on a US bank or international money order. Resetting of ad constitutes a new advertisement at first insertion rate. Camera ready copy from laser printers, photo typesetters, etc., are acceptable. Dot matrix, daisy wheel, typewriter output not accepted. Inquire for rates for larger ads if required. Deadline is eight weeks prior to publication date. Mail to:

The Computer Journal
Market Place
PO Box 12
S. Plainfield NJ 07080-0012 USA

CP/M SOFTWARE

100 page Public Domain Catalog, $8.50 plus $1.50 shipping and handling. New Digital Research CP/M 2.2 manual, $19.95 plus $3.00 shipping and handling. Also, MS/PC-DOS Software. Disk Copying, including AMSTRAD. Send self addressed, stamped envelope for free Flyer, Catalog $1.00

Elliam Associates
Box 2664
Atascadero, CA 93423
805-466-8440

For: CP/M Users with a sense of humor

WHILE YOU WERE OUT

Mr Bradley

of: Small Computer Support

Address: 24 East Cedar Street
Newington, CT 06111

[ ] called for you
[x] stopped by
[x] wants to hear from you

Message:

Remember Pieces of 8? It's back, better than ever as Eight Bits & Change, a bimonthly newsletter filled with humor, tutorials, graphics and fine technical articles. Only $15 per year in the U.S. ($18 in Canada and $21 foreign.) Subscribe today! Satisfaction guaranteed!

Z-System Software Update Service

Provides Z-System public domain software by mail.

Regular Subscription Service
Z3COM Package of over 1.5 MB of COM files
Z3HELP Package with over 1.3 MB of online documentation
Z-SUS Programmers Pack, 8 disks full
Z-SUS Word Processing Toolkit
And More!

For catalog on disk, send $2.00 ($4.00 outside North America) and your computer format to:

Sage Microsystems East
1435 Centre Street
Newton Centre MA 02159-2469

Real Computing

X-10 Revisited, Mach, Minix, and Desqview/X

By Rick Rodman

X-10 Revisited

In issue #48 I wrote glowingly of the X-10 Powerhouse computer interface, known as the CP-290, that "any X-10 commands which come across the line are converted to status messages that the computer can see". Mike Morris writes: "I have a CP-290, and mine does not produce these messages when I push a button on a manual controller."

After some experimentation and consulting with the X-10 gurus at Home Control Concepts, I have to admit Mike is right. The CP-290 does report to the computer when buttons are pressed on it itself, but it doesn't report codes that come over the wire. It is a send-only, unidirectional device.

Actually, the plot's even thicker than that. Manual controllers, like the wall switch modules, don't send codes at all, they only receive them. There is no way for a central controller to know whether a user has manually turned a light on or not.

There is another computer interface available, which is called the TW-523. This device has an optoisolated TTL-level interface which can be operated through parallel input and output bits. It's "dumber" than the CP-290—it has no battery backup or timers—but it's bi-directional: the computer can send or receive any X-10 code over the power lines. Remember, the manual buttons on the wall switch units don't send any code. However, you can receive codes sent by other controllers or by motion detectors.

If you're planning a system controlled by a dedicated computer, the TW-523 sounds like the way to go. It's available from Home Control Concepts for $30. It's also available as part of a Powerline Interface Kit for $69, which includes a C library and sample code as well as a cable for connection to a PC parallel or serial port.

HCC also has available some wireless motion detectors. These detectors send an on-code when motion is detected, and an off-code when motion stops, through the same base unit used by the wireless hand control. Because they're wireless, you can put them anywhere—out in the yard, garage, parapet, moat, wherever. Because they send a code, you can not only turn on a light or sprinkler system, you can also detect the code in your TW-523 and activate a voice synthesizer or camcorder, or switch speakers connected to the stereo, or anything else.

Mike Morris suggests some neat applications for X-10: "I want to be able to turn on the sprinklers when the ground gets dry or if the burglar is leaving, but not the day after a rainstorm. I want to run the dishwasher after everybody goes to bed, then turn the water heater off but back on an hour before I get up. And divert the clothes dryer exhaust into the furnace air intake when appropriate so the furnace grabs the preheated humidified air."

Mike also provided some "from the trenches" information on X-10: "The X-10 signal can have problems 'jumping' across the two halves of a 220 volt household circuit. I ended up acquiring a capacitor of the proper value, installing it in a 220v plug and plugging it in. 'One of these days' I'll install the Leviton capacitor module in the main electrical panel, but for now, it works."

If you're planning a system controlled by a dedicated computer, the TW-523 sounds like the way to go.

He also points out that some X-10 lamp modules "forget" their brightness setting and will "creep" up or down over several hours, and suggests "refreshing" their settings periodically. (That'll also solve the problem of people using the manual buttons—and staying too long in the bathroom, too!) He points out that the CP-290 doesn't keep accurate time, losing about an hour per month. Mine has forgotten its house code a couple of times. "The battery compartment in the CP-290 does not have a barrier between the battery and the circuit board. Mine leaked and ate away a few traces, which I repaired with a small soldering iron and 30 gauge wire."

Not only X-10, but the whole field of home automation is booming in popularity these days. The CE Bus, a fully-bi-directional superset of X-10, is coming. I'll venture a guess that the reasons for this sudden interest are (a) the increase in crime, especially senseless violence and vandalism, (b) higher energy costs, and (c) it's inexpensive and loads of fun!

Mach on the PC-532

Carnegie-Mellon has released a "Micro-Kernel" for Mach which contains no AT&T code. If you have ftp access (which I don't), you can get it and play with it. It's called a "micro-kernel" because it includes no I/O or utilities. A "UX server" task, made apparently of pieces of Unix, is run to provide the missing capabilities. Some

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.

Reader, from page 47

applications."

Did it fit?—Ed.

I think the author of the previous letter said it well. I'm sure what I look for in a magazine, but I think I recognize it when I see it. I sure did like MicroCornucopia. *TCJ and* CCI *are my replacements for it, but TCJ wins out with its Forth coverage. I liked* DDJ *in the early days, loved* Kilobaud *and the early* Byte. Kilobaud *is dead and* Byte *might as well be. The common thread seems to be* hardware, *and hardware that can be built and fooled around with on a limited budget. Obviously I like software too. Maybe we are a dying breed. I hope we with an interest in both hardware and software are not. I think magazines such as* Popular Electronics, Radio-Electronics, *whatever, even though they tend to be criticized for poor technical content, suggest that there is a good target audience out there for TCJ. If so, how do we reach them?—Frank Sergeant, Contributing Editor.*

———

I am glad I made it to the Trenton Festival this year: besides meeting you, Al Hawley, Ian Cottrell, Howard Goldstein, Bruce Morgen and Harold Bower for the first time, it broke the sense of isolation I sometimes feel in this remote corner. I found a lot of commonality with what Bower is engaged in and hope to set up some exchanges with him. Looking forward to further developments on banked systems and new Z-180/280 boards!

L.V.H. Mill Creek, WA

The Trenton Festival is a great way for CP/Mers to get together, and it was a pleasure meeting you, too. There were some unfortunate oversights on the part of the sponsoring clubs but we got on pretty well despite that. Talk is that there will be a special banquet for 8-bitters next year. Bob Dean and I have been appointed joint ring-leaders to get it organized. Be looking for more information in the next few months.—Ed.

———

Enclosed is a check for $25 US. The reason it is more than you asked for is to allow for the extra aggravation you may experience. I recently learned American publishers are running up against our Customs bureaucrats....

I see future use for old CP/M machines dedicated to monitoring and controlling homes and offices such as Jay explained in his articles. I would be particularly interested in more articles, particularly concerning building systems control/monitoring such as heating, ventilation and security checking. It would be beneficial to call up your place during a winter storm and check out the furnace or verify against unauthorized entry before calling the police.

I would like to see an article dedicated to RS232 standards, common usage with peripherals and those inevitable deviations from standard, as Art Carlson asked about earlier.

Keep up the good work, we readers do care. Some of us just do not communicate the fact often enough.

P.C. Edmonton AB

Thank you! I haven't any problems yet, but will be watching.

I like your idea of CP/M boxes to control the home. But don't limit yourself to using a general purpose computer. Many of the embedded controller types who frequent these parts can cook up quite a system with an F68HC11, 8031, Z8 or similar.

Thanks for your letter. You are right, the authors need your feedback and so do I!—Ed.

———

I am operating under TurboDOS. Any chance of a TDOS article?

H.R. Philadelphia PA

Good question! Requests have also been received for articles on CDOS and S-100. S-100 Buss dropped S-100 support, of all things! Any takers?—Ed.

———

My company is spearheading an effort to develop a SCSI host adapter for the Zenith Z-100 computer and I noticed that some of your back-issues have articles relative to this project. Please send issues number....

P.F.H. New Port Richey FL

Your issues are on their way. Keep us posted on your progress with the Z-100 project. Sounds like something we'd be interested in.—Ed.

———

I have a Kaypro 4+88. Is there anything past MS-DOS 1.25 for this machine? Also, has anyone replaced a Z80 with a Z280?

P.B. Broadview IL

I'm afraid you can't pop a Z280 in place of a Z80. Though the '280 is software compatible, it is not compatible on the hardware level. Have you been following the YASBEC articles?—Ed.

———

Got the magazine a few days ago. I'm impressed. It has a good format and the articles are well done, informative and useful. I'm quite new to CP/M and the Z-System so some things were just a little over my head, but that just means I need to work with the computer more to gain a good level of knowledge. I have found the CP/M group here in Portland.

As for the handwritten letter, it was a nice change from the normal mass produced standard form letter.

R.D.W. Portland OR

Thanks for the compliments! Yes, we are blessed with great authors. I like your attitude about learning more to reach the level of the articles rather than pulling the articles down. Can't learn that much.

Note to others: Richard noticed that I have never bought a typewriter. I wrote a handwritten letter to him. One day I may spring for a Selectric though I really prefer to stay with state-of-the-art technology, like 8-bit machines....—Ed.

———

How about some material on the *old* CP/M Kaypro II/IV's? I may even have some to contribute!

F.G. East Greenville PA

You're on, Frank! Let's see what you have in mind.—Ed.

———

I'd like to see articles on X-10 software, hardware and interfacing; sound generators and voice synthesizers.

R.D. Longueuil QU

I agree and am hoping to see some good submissions.—Ed.

———

My primary attraction to *TCJ* is as a "last bastion of 8-bit computing," much as Jay Sage has represented on the Internet and in his announcements on Z-Node 3. Over the course of the year, however, I have witnessed the departure of Bridger Mitchell's Advanced CP/M column and was disappointed to see nothing further along the lines of the article on available S-100 boards. The increasing focus on Z-System and embedded controllers reminds me all too much of *MicroC's* abandonment of "us poor CP/M'ers," who were grateful for even such simple things as current vendor listings.

See Reader, page 52

PMATE/ZMATE MACROS

5. Downloading Earlier Columns and A Sort Macro

By Clif Kinne

Since each of these columns presumes that the reader has access to preceding columns of the series, new subscribers are at a severe disadvantage. To alleviate this inconvenience, I shall post each column to Jay Sage's ZNode BBS a couple of months after you receive it in the mail. This delay will allow me to incorporate any correction and clarification that appears needed. (Those who have worked their way through earlier columns know that I need some such escape valve.)

Downloading Columns

As of now there are just two files to be downloaded:

PZMCOLS.ZIP: PMZMCOLS, an acronym for PMATE/ZMATE Columns, is a "zip file" library of 4 files,- one for each of the first 4 columns:

PZMCOL01.RV1
PZMCOL02.RV1
PZMCOL03.RV1
PZMCOL04.RV1

The extension, RV1, indicates that the file has been revised once since it appeared in *TCJ*. PZMCOLS.ZIP is about 32 kilobytes long, whereas the individual file lengths before compression total about 82 kilobytes.

PZMACS.MAT: This is a file of all macros presented in the first four columns, each in executable form without comments. Since, in this form, they use relatively few bytes, they are not compressed. This file may be downloaded and individual macros put to immediate use with a minimum of tailoring to your system and taste.

The downloading process.

Jay Sage's Z-Node BBS is at (617) 965-3552. Although this BBS caters to the Z-System and CP/M, the files are equally accessible to DOS.

If you are running on CP/M or the Z-System, you will need UNZIP.COM to decompress the files. If you don't have it, you should get it. Jay has it on the Z-Node. It is in the library file, UNZIP15.LBR, whose members are crunched, so you will need UNCR.COM also.

If you are running DOS, you will want PKUNZIP.EXE, which is in the self-unzipping library file, PKZ110.EXE, which, in turn, is available on most DOS BBS's.

If you do have occasion to download these files from Jay's Z-Node, please take the occasion to leave me a brief message. If nothing else, what version of PMATE are you running: ZMATE, PCMATE, MATE, or more than one?

A SORT MACRO.

Eight years ago this August, *Lifelines* published a macro that prepares a sorted, columnized disk directory. The sort part of the macro was submitted by Ron Finley of Harley, Oregon. Over the years I have greatly embellished and refined that DIR macro, but never modified the sort part. Now, thinking about offering them in this column, I thought I might spread them over several issues and start with the sort macro in this issue.

Listing 1.
The basic Sort Macro.

```
^X'     ;Sort                                         82 bytes

;       FUNCTIONAL SPECIFICATION:

;       Sorts a column (unsorted list, UL) of text items in
;       one buffer into alphabetical order in a second
;       buffer (sorted list, SL).

;       The inner loop moves the SL cursor successively
;       closer to the alphabetical target for the next item
;       in the UL, each move being half the distance of the
;       preceding, so that, on the last move of 1 line, it
;       winds up just above or just below where it should
;       be.

;       The outer loop corrects that 1-line uncertainty as
;       necessary before going for the next item in the UL.

;       VARIABLES USED: V0, the no. of items in SL.
;                       V3, the no. of lines moved on the
;               current pass throught the inner loop.  V3 is
;               cut essentially in half after each pass.

;       BUFFERS USED:   B4 holds the unsorted list, UL.
;                       B2 holds the sorted list, SL.

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

When I went over the macro with some care, it seemed to be difficult to justify some of the tortuous steps taken to achieve its flawless performance. So I asked myself what I should do if starting from scratch. The end result is the macro depicted in *Listing 1*. This consumes 82 bytes, compared to 145 bytes in the original. It seems to work as flawlessly as the

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

original, and most of the steps I find to be quite plausible. I hope you do. The one exception is on line 14, the incrementing of V3 before dividing by 2. I can only justify this empirically. Maybe one of you can tell us how we could have anticipated that this was an appropriate or necessary step.

On my 12 Mhz AT, this seems to take a second for every 16 items in the unsorted list. This is not terribly fast, but is very acceptable for an average 50- to 60-item directory. On my 4 Mhz CP/M machine, it is 5 or 10 times slower. In neither case, did I recreate the directory every time I displayed it. Instead, I saved it to disk and simply redisplayed it most of the times. In fact, with a fairly large directory in CP/M, I often made a few patches by hand before creating a new one.

```
;          USAGE:  The unsorted list must be in B4 when this
;                  is called.  Then it may be called from any
;                  buffer.
;          CODE:
.^S                ;Save the environment.                    1
0V0                ;Clear V0 (No. of items in SL).           2
B2K                ;Clear destination buffer, B2.            3

[                  ;Begin outer loop through UL.             4
  B4E              ;Go to UL.                                5
  A                ;To top .                                 6
  @T=0_            ;Terminate if UL is empty.                7
  B2N              ;Move first item in UL to SL.             8
  VA0              ;Increment no. of items in SL.            9
  B2E              ;Go to SL.                               10
  A                ;To top.                                 11
  @0V3             ;Initialize V3 to the no. of items in SL.12

  [                ;Begin loop to put top item of UL in its 13
                   ; alphabetical place in SL.
    @3+1/2V3       ;Increment previous V3, then halve it.   14
    @H^A@4$<0      ;IF item at cursor < top item in UL,     15
      {@3L         ;  THEN move down.                       16
      }{-@3L}      ;  ELSE move up;                         17
    @3=1           ;IF V3 is 1 at this point, you are done. 18
  ]                ;ELSE loop again.                        19

  @H^A@4$<0        ;IF item at cursor < item in UL,         20
  {L}              ;THEN move down (to correct uncertainty).21
]                                                           22

.^R                ;Restore the environment.                23
B2E                ;Return to SL.  This step is optional.   24
```

For purposes of calling this in future columns, I shall have to give this a name, and shall simply use the one I have been using for years, namely, .'.

Speedier sorting of a partially sorted file.

Having gone this far, I worked out a modification that cuts the time to 40% if the UL is mostly sorted with a few random additions at the end of the list. So if you are using a DIR sorting utility, the macro will take advantage the presorting. My deadline is upon me, so I shall simply make *Listing* 2 the code for the lines of *Listing 1* which are changed.

The first line is line 11 from *Listing 1,* with the preceding semicolon to indicate that it is to be omitted. It is replaced by the following lines, 11a, 11b, and 11c. The only other change is the addition of line 21a following line 21.

As you can see, the new code simply checks first to see if the next UL item belongs at the bottom of SL. If it does, the inner loop is never entered. Otherwise, the inner loop code is executed exactly as in *Listing 1.*

I wonder how many hours I would have saved if I had taken the time to work these things out 8 years ago. On second thought, I'm happier not knowing.●

Listing 2.
Code modifications to Listing 1 to speed sorting a partly presorted list.

```
; A              ;To top.  (Remove original line 11)
  Z-L            ;Move to last item in SL.               11a
  @H^A@4$>0      ;IF item at cursor > top item in UL,    11b
  {              ;THEN execute steps 12-21 of Listing 1. 11c
  }{L}           ;ELSE move to line below last item.     21a
]                ;and return to start of outer loop      22
                 ; to put top item in UL here.
```

Reader, from page 50

I plan to build an S-100 board containing a real-time clock and work extensively with BDS C and Aztec C under CDOS. If articles on either of these topics would interest you, please let me know.

J.J.G. Lexington MA

We have been working more with embedded control topics as they have proven quite popular with the readers. Please note, however, that I added nearly 30 pages to provide room for the new material without denying our roots, the CP/M'er. I am as concerned about abandoning our charter as anyone. For the record, we did not drop Bridger Mitchell. He has been very involved in his professional work and has promised to return when he can. I look forward to his next article.

There are two things needed to assure support from any magazine: articles to publish and readers to buy the result. We have the readers (can always use more!) and I am interested in the articles you suggest. Frankly, I am rather curious why the authors and readers of the old S-100 *Buss haven't looked this way.*

Sometimes a publisher sees greener pastures and moves beyond where the readers want to be. Let me know if I do this to you.—Ed.

Is W.H. Laidley suggesting, indirectly, that real power protection is only available via a rotary converter, i.e. an AC motor driving an AC generator, preferably with a big flywheel to smooth over hiccoughs?

A.S.M. Bois-Colombes, France

No, Mr. Laidley's article in issue 49 addressed two issues. He takes issue with using the ground plane to absorb surges as it is poorly isolated from signal ground in most systems. This would divert high energy away from the power supply, a relatively cheap and robust unit, and toward the TTL-level ports which are much more expensive and delicate. He also has concern with using MOVs in power protection and gives two principle reasons: An MOV breaks down a little at a time with every surge it absorbs, yet never warns that its protection is failing, and MOVs are subject to thermal runaway. Layman's talk for this is "catching on fire."

Implementing Mr. Laidley's recommendations involves using devices other than MOVs and avoiding use of the ground plane. His firm, Zero Surge *of Bernardsville NJ, sells units that meet these criteria. The devices are neither cheap nor trivial (they cost about $200) but from my experience so far, they are effective. After losing issue 49 to a power surge (the hard disk FAT was scrambled), I got one of the units for Amanda, the typesetter.—Ed.*

You have a great magazine. I've been playing with com-

See Reader, page 57

A Home Heating & Lighting Controller, Part 3

The Computer Hardware

By Jay Sage

In the two previous installments on the embedded controller that runs the electrical and heating systems in my house, I described first the history behind its development and the control strategy it applies to managing the heating system and then the electrical interface between the controller and the 120 V electrical system. Since *TCJ* is, after all, a magazine about computers, it seems high time that I talk about the controller itself, and that is what I will do now.

There is no sense going into the full schematics of the circuit. This controller was designed and built ten years ago, and integrated circuit technology has advanced tremendously since that time. Besides, to be completely honest, I cannot find a full set of up-to-date schematics. Quite a few changes were introduced along the way, and their documentation is embarrassingly sketchy. In addition, after all these years, I no longer remember in detail how each part of the circuit was intended to function.

It is interesting from an historical perspective to see how such a controller was implemented ten years ago.

The above notwithstanding, my main reason for not going into all the details is that, were I going to implement the functionality of this controller today, I would not use the same circuit design. Nevertheless, there are some useful things I can say about the design that, I think, still apply today. It is also interesting perhaps from an historical perspective to see how such a controller was implemented ten years ago.

Overall Description of the Hardware

Most of the controller is housed in a small wooden box (a previous hobby was woodworking) mounted on the wall of our hallway. *Figure 1* is a drawing that I made for a talk presented to the Lincoln Laboratory computer club just after I joined the lab in 1982. The box is about 9 inches high and 7.5 inches wide. It is open on the bottom, where a reset button and a power switch can be accessed by reaching under the box. The top is also open, but a metal grill is installed to prevent objects from falling in. This arrangement provides excellent cooling!

The operator provides input to the controller via a hexadecimal keypad. Numbers are entered directly, while functions are invoked by holding down a small pushbutton switch in the lower right corner while a number key is pressed. The pushbutton switch acts, if you will, as a shift key.

Information from the controller to the operator is provided by a 16-digit 7-segment LED display. This unit was intended to display numbers in calculator applications, but with some ingenuity I was able to generate reasonable approximations to many letters as well. In *Figure 1,* for example, the word "ALL" is displayed. Since some letters cannot be formed (e.g., G, J, K, M, and T), a lot of thought had to go into devising appropriate display messages!

Inside the box are two boards with the main computer circuitry. The first board was to contain the actual computer, while the second board would take care of analog signal conditioning and analog-to-digital conversion. Later, some additions to the main computer overflowed onto the second board as well.

Wirewrap sockets installed in prototyping "perf" boards were used for mounting all components, both integrated circuits and passive devices (mounted in carriers). A Vector Slit-n-Wrap tool was used to wirewrap the circuits. This tool really expedites such work because it slits the special wire as it wraps it onto the wirewrapping posts. The wire is cut after it is wrapped, and one can daisy-chain the connections when lots of pins are bussed together, as happens often in computer circuits.

Some additional hardware is mounted near the ceiling in the basement below the control panel. The power supply, which we will describe in more detail later, is there. So is the board with the relay drivers that we described last time. All the high-frequency circuitry is on the boards in the control panel, but the low-frequency and interface circuitry was kept out of the control panel to minimize its size and the number of wires that would have to go to and from it.

When one anticipates high volume production, one avoids costly parts, even at the expense of more complicated designs and software.

The General Design Philosophy

Before delving into the specifics of the circuits, I would like to say something about the general approach I took with the design. When cost is an issue—as when one anticipates high volume production—one tries to avoid costly parts, even at the expense of more complicated designs and, especially, more complicated software. Since I anticipated building only one of these controllers and since I wanted to spend my programming time on the functional software and not on low-level interface issues, whenever possible I chose high-level chips that performed low-level tasks in the hardware. Spending $20 on a fancy new IC was better than spending two weeks writing and debugging software. There will be several examples of this discussed below.

Some Circuit Details

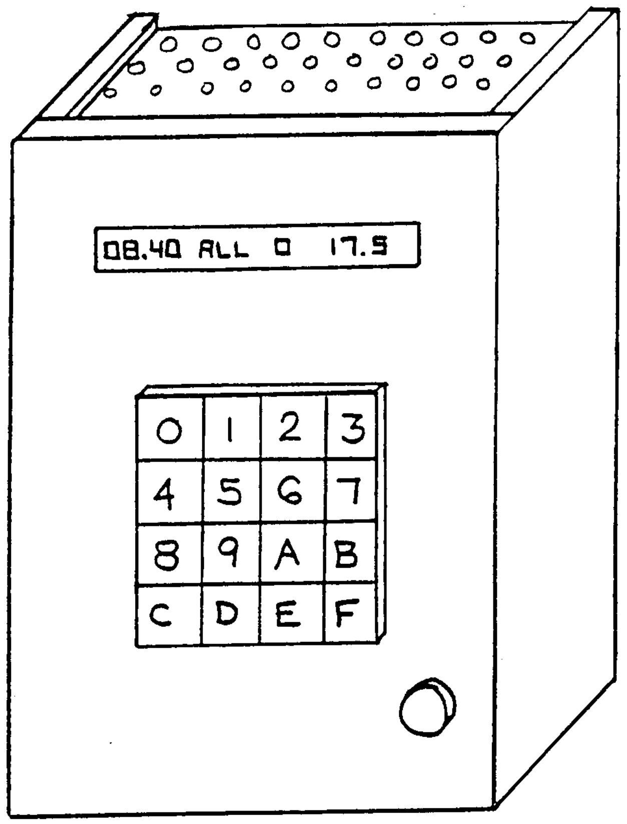
The CPU

A highly abstracted schematic of the main bus of the controller is shown in *Figure* 2. The controller was designed around the state of the art CPU at the time, the Intel 8085. By multiplexing its data and address signals onto the same pins, the 8085 freed up enough pins to implement additional I/O functions on the chip. Among these are a single-bit input port and a single-bit output port. These can be used to implement a bidirectional serial interface. I made no use of them, however.

More importantly, the 8085 contains an integral prioritized interrupt controller and four vectored interrupt inputs ('vectored' because they force execution of specific code addresses and do not require an external controller to jam instructions onto the bus as with the standard interrupt input, also present on the chip). These I used to great advantage.

One interrupt line is called TRAP and is non-maskable (always active). Three maskable interrupt pins — named RST5.5, RST6.5, and RST7.5 — are controlled by a special CPU register via 8085-specific opcodes: SIM (Set Interrupt Mask) and RIM (Read Interrupt Mask). These interrupts generate calls to locations in the base page of memory, just like the standard RST (Restart) opcodes. The addresses called are midway between those called by the RST instructions — hence the names.

Figure 1.



To facilitate the design of very small controllers, Intel provided a whole family of chips designed to interface easily to the 8085. These chips include on-chip circuitry to demux the data and address buses. I used several chips from this family, but I also used conventional microprocessor bus ICs. Thus I had to provide external bus demuxing circuitry.

Memory

First, the computer obviously needed memory. This took two forms in the controller: RAM and ROM. The RAM was provided by the 8155 multifunction companion chip to the 8085. Within its 40-pin package, the 8155 provides 256 bytes of RAM along with three parallel I/O ports (a total of 22 bits), and a counter/timer. Two of these chips were used. One counter counts the output from a 60 Hz timekeeper circuit and produces a pulse each second. This signal is routed to the RST7.5 interrupt, where it initiates the main computation cycle performed by the controller.

The output of the one-second timer drives the counter in the second 8155. I can't remember what I intended to use that counter for. In my original schematic, it is shown connected to the RST6.5 interrupt, but there is no code for that interrupt (and presumably the line is not connected).

Only a few of the parallel port lines are used, and one 8155 chip would have been enough. The second one was included mainly for the additional RAM. The principal use of the parallel port lines is to control the relay drivers. As discussed last time, this requires six lines. One other line seems to have been used as a direct inhibit (blanking) on the display (I don't remember why).

In a way I am amazed that I thought that a mere 512 bytes of RAM would be enough for the entire controller (and, just think, today they say that 4,000,000 bytes will barely get you by under Windows!). Later I decided that I really wanted more RAM so that the controller could log environmental data (it was that data that revealed to me how incredibly accurately the controller was following the programmed temperature profiles — much better than my theory had predicted).

I added four 8185 1 kB static RAM chips. These chips were developed as part of the 8085 family and included the bus interface. When I first built the controller they cost something like $50 each, and that was more than I was willing to pay. Had I included even one of them from the beginning, I would have used only a single 8155.

In the early 1980s, EPROMs were still very expensive. The 8755, an EPROM dual to the 8155 RAM, was unthinkably expensive at the time. Even the state-of-the-art 2716 2k-by-8 EPROMs cost more than $50 (today they are, I believe, well under $5), so I selected the smaller 2758. These were, it was said, 2716s in which one of the two 1k banks was defective, leaving a 1k-by-8 memory. Ironically, as IC production technology improved, the yield on 2716s became so high that there were few defective chips from which to make 2758s, and the latter actually became more expensive than full 2716s! Just in case my program code eventually grew to more then 1 kB, I included a second ROM socket. Today the controller has both sockets filled with 2732 4k-by-8 EPROMS! The general rule seems to be that one NEVER has enough memory.

Keyboard and Display

Designing the logic to handle the hex keypad and LED display would have been a considerable task were it not for the Intel 8279 keyboard/display controller chip. This chip had everything I needed.

The keypad presented no challenge to the 8279, which was designed to scan a full alphanumeric keyboard. Three output scan lines (decoded to eight) and eight input return lines allow the chip to handle an 8-by-8 matrix of keys. The hex keypad was only 4-by-4. The 8279 also has inputs for shift and control keys. My pushbutton 'function' key was wired to the shift input. When the 8279 detects and verifies a pressed key, it asserts an interrupt signal, which I wired to RST5.5 input on the 8085.

There were actually two other input keys that I have not mentioned before. A GE low-voltage switch is mounted by each of the two main entrance doors of our house. One position on the switch is marked 'home', the other, 'away'. The controller is capable of modifying its schedule based on whether we are at home or away, and these two switches are used to convey that information to the controller. They drive optical isolators, whose outputs are read by two additional return lines on the 8279.

The LED display is handled with equal facility. While only three scan lines are used for the keyboard, a fourth is present for use with the display. When these lines are decoded, 16 digits can be addressed, a perfect match to the display I chose. Eight driver lines control the seven segments of the digit display plus the decimal point.

Analog-to-Digital Conversion

The last major functional block of the controller is the analog signal processing unit. For the A/D converter, I again opted for a sophisticated IC that would handle as much of the work as possible. The National Semiconductor ADC0816 was the state-of-the-art chip at the time. It supports 16 analog channels and performs successive-approximation ratiometric conversions in 100 microseconds. ('Ratiometric' means that values are scaled to a reference signal, for which I used the power supply that biases the temperature-sensing thermistors. This makes the results independent of that supply voltage.)

The ADC0816 also includes a full microprocessor interface, so that it could be accessed essentially like a RAM chip. By writing to the chip, the channel could be selected and a conversion initiated. The result is obtained by reading from the chip. When a conversion is complete, the ADC0816 asserts a signal on a control line. This could be used to initiate an interrupt, but, since the controller had nothing else to do in the meantime, it was easier to execute a software delay loop after starting a conversion.

The Real-Time Clock

The above discussion covers all of the circuitry on the main boards in the control panel. I would now like to describe two of the additional circuits mounted remotely: the real-time clock and the uninterruptible power supply.

As I mentioned earlier, the time is derived by applying a 60 Hz signal to the counter in one of the 8155 ICs. This 60 Hz signal can come from one of two sources.

The main source is the 60 Hz line. The electric company controls this frequency very carefully. It appears, in fact, that when the frequency is low for some period of time, it is later raised slighter until to total number of cycles is correct. This is why electric clocks keep such accurate time, unless there is a power failure.

I took the AC from the secondary of a bell transformer and applied it to the input of an optical isolator. The LED in the isolator serves as a rectifier, so that the output on the other side of the isolator is a 60 Hz pulse train. Two steps are then taken to make sure that this signal is clean (does not contain many spikes at the transition threshold). First, the signal is passed through a Schmitt trigger, and then it drives a nonretriggerable monostable set to produce a 5 ms pulse. This signal serves as one input to a two-input multiplexer.

The other clock signal, which is applied to the other input of the mux, comes from a crystal-stabilized clock chip. Using a crystal at the color subcarrier frequency of a television signal, this chip produces a 60 Hz pulse train. This signal, however, is far less accurate than the one derived from the AC line.

How do we determine which signal to use? Well, the signal derived from the AC line also goes to a retriggerable monostable which is adjusted to a 20 to 25 ms pulse. So long as the AC power is present, pulses come at intervals of 16.7 ms, and this monostable is constantly triggered. As soon as a single cycle of the 60 Hz AC line is missed, however, this output drops. This does two things. First, it causes the multiplexer to send the crystal timebase signal to the 8155 counter. It also serves as a power-fail signal and is connected to the TRAP interrupt. When we discuss the controller's software next time, we will describe how the controller uses the knowledge that the power has failed.

The UPS

I don't think I really have to say that maintaining power to the controller is of the utmost necessity. Thus, an uninterruptible power supply was essential.

Basically, the controller runs off a 5 amp-hour, nominally 8 V Gates sealed lead-acid battery. This uses similar technology to that in a car battery, but the dangerous gases are not allowed to escape from the cell. The second computer board contains a regulator IC that drops the voltage to an accurate 5 V. The battery was designed to keep the controller running for at least 8 hours.

Meanwhile, so long as the AC power is available, the battery is charged by a power supply. A small transformer, a bridge rectifier chip, and an adjustable voltage regulator IC generate a roughly 10.2 V signal. This signal passes through a diode on its way to the battery so that there is no chance that the battery could discharge through the power supply during a power failure. The variable regulator is adjusted to produce a voltage of 9.2 V on the battery side of the diode, the level that Gates recommends for continuous charging.

That covers what I intend to say about the controller circuitry and hardware. Next time we will take a look at the software.●

Figure 2.

```
       Schematic of the main bus of the controller

CPU    ....EPROM....   ..RAM-I/O...      ADD'L RAM

8085    27xxA   27xxB   8155A  8155B        4 x 8185
|        |       |       |      |        |   |   |   |
+--------+-------+------++------+--------+---+---+---+
|                       |
8279                  ADC0816

KEYBOARD/DISPLAY       16-CHANNEL A/D
CONTROLLER             CONVERTER
```

Assembler Programming, from page 24

SCR_LD updates the value field in each screen image for each configurable item in the target configuration page, then stores a copy of the first *Menu Data Structure (MDS)* in a working buffer (MENLST:) in ZCNFG.

A few details have been omitted in that definition! The CFG file was originally assembled, then linked to make a binary image file. That means that the addresses in the file start at some value which we will call ORGCFG. You may be sure it is not now loaded at that address! Therefore, the pointers in MDS are all wrong by a constant which is the difference between (OVRLAY) and ORGCFG. This difference is called a relocation constant; it is calculated in SCR_LD stored for future use at RELOCC:. The relocation constant is added to any address from the CFG file before it is used. The MDS was relocated during the copy operation.

Recall that there is an MDS for each menu in the CFG file. Each MDS is part of a doubly linked set of records. The first two words in each MDS are pointers to the preceding and following MDS in the list. Each MDS also contains pointers to its related SCREEN IMAGE, CASE TABLE, and HELP SCREEN. These pointers allow ZCNFG to locate and work with any data in the CFG file.

There is another detail whose omission you may have noticed. The items in the target configuration block are all binary, and must be converted to any one of a number of ASCII formats in the screen image. One of the entries in each CASE TABLE record specifies the conversion that must be made. So part of the update process mentioned above is to call the appropriate conversion routine. User input during configuration is always ASCII (that's how keyboards work!); that input is converted to binary, stored in the target configuration page, then converted to screen format and transferred to the screen image. The addresses for these conversion routines are stored in a table (FNTBLE:). There two symbols associated with each function in the table. The first symbol is IN_FNx, a keyboard input and conversion function; the second is LD_FNx, which loads a screen image field from a configuration item. 'x' is a function number. See *Listing 3*.

ZCNFG is finally ready to go to work for you!

Listing 5.

```
                PROGRAM EXIT

;Normal exit after changes are made and need to be
;saved. Assumes that tgtbuf has been updated before
;writing the first block back to the file.

EXIT:   LD      A,(Z3INST)
        OR      A               ;allow Z3 installation?
        JR      Z,EXIT1         ;no, if logical false
        LD      HL,(Z3ENV)      ;install current ENV address
        LD      (TGTBUF+Z3EOFF),HL

EXIT1:  LD      A,(FCB_CR)      ;set current record for writing
        LD      (FCB+CREC),A    ; back the configuration block
        LD      DE,TGTBUF
        CALL    SETDMA
        LD      DE,FCB          ;fcb pointer
        CALL    WRREC           ;write first 128 byte record
        EX      DE,HL           ;preserve fcb pointer
        LD      DE,TGTBUF+80H
        CALL    SETDMA
        EX      DE,HL
        CALL    WRREC           ;write second record
        LD      A,(TGTDU)
        CALL    CLSFIL          ;close the file
        CALL    Z3CLS           ;clear the screen

;Here to exit without saving current changes. This is
;effectively an abort. Assumes that the saved stack pointer
;points to a safe execution address. This is the case when
;the program is executed as a normal .com file.
QUIT:   LD      A,(Z3MSGF)
        OR      A               ;if ZCPR3 present,
        CALL    NZ,DINIT        ; deinit terminal
        LD      SP,(STACK)      ;restore callers stack
        RET                     ;return to CCP
```

THE MAIN PROGRAM

The core of ZCNFG is only about 20 lines of code! It starts at the label SETOPT: and ends with the instruction JR SETOPT. This looks like an infinite loop! Here is the sequence of events in this code:

1. display the current screen image (menu)
2. display the ZCNFG commands and a user prompt
3. get and display user input (menu selection)
4. CALL MCASE to perform the action implied by the user input
5. If MCASE returns an error flag go to 3, else start again at step 1.

The secret is in step 4, the call to MCASE, which:

1. *searches the current case table for an item which matches that input by the user.* If found, jumps to the appropriate service routine from the list of routine addresses in the table labeled FNTBLE: If not found, continue at step 2.
2. *searches the case table built into ZCNFG.* This table is located at label CTLCS0:. These are for the functions shown at the bottom of *figure 1*, "ZCNFG INSTALLATION CONTROL". Notice that one of the functions is to exit the program, with or without saving changes. As in step 2, jump to the appropriate service routine or if no match for user input is found, continue at 3.
3. *Returns an error flag.*

In all cases except those involving exit, a return to the caller of MCASE occurs. Refer to *Listing 4*.

The Program Exit

There are two ways to leave ZCNFG: with and without saving changes. The routine at EXIT: writes the current configuration page back into the target file. This page contains any updates made during the ZCNFG session. After the write, the target file is closed, the screen is cleared and the QUIT: routine takes over. QUIT deinitializes the terminal (if required), and performs one final critical action. It restores the Stack Pointer to that of the CCP and does a RET instruction for a silent return to the CCP. ZCNFG is finished. See *Listing 5*.

Subroutines

You will observe that most of the work done by ZCNFG is done in calls to subroutines. And the subroutines call more subroutines! Those that are really unique to the flow of logic in the program are a part of the SUBROUTINE section near the end of the source file. Lower level subroutines which are simple functions are in a separate file, CFGSUBS.Z80. Still other functions are commonly used and come from such standard rel libraries as Z3LIB, VLIB, and SYSLIB. ZCNFG also uses a special rel library, CFGLIB, which contains modified filespec scanning routines from ZCPR3.

Some subroutines are only called once. Why not just include them in-line? One reason is to keep main sections of code short, preferably less than one page long. Intelligent comments are required to keep the reader (you, a year or two from now) from getting lost. A second reason is illustrated by the MCASE routine; it contains some very tricky code that would become even more treacherous as in-line code. Debugging and maintenance are usually easier for a subroutine than for equivalent in-line code.

Data Section

Initialized data comprises DB or DW statements whose value is determined during assembly and linking. Uninitialized data comprises DS statements and may also result from ORG statements. An area of uninitialized data cannot generally be depended on to be filled with any particular value; this is data space to be used during program execution. The buffer in which the configuration page is stored is a typical example.

The program structure in *figure* 2 implies that uninitialized data should be last in the program. Linkers like ZML will produce a COM file which does not include DS areas at the end of the code image. That makes for a shorter file to store and load. But beware! The only way to be really sure that this really happens is to declare DSEG for all DS areas (and be sure to switch back to CSEG for non-DS areas). ZMLs default linking strategy is to place all DSEG after CSEG in the COM file; that will be last if no COMMON segments have been declared. In such a case, terminal DS areas will not be included in the COM file. That is also why you cannot depend on the values initially present in DS areas; they are undefined unless you have filled memory with a known value before loading the COM file for execution! If this all seems obscure, just remember to habitually follow the DS last strategy. It will eventually make sense.

Auf Wiedersehen

We have reviewed the overall organization of a typical AL program, using ZCNFG as an (imperfect) example. Along the way, some of the strategy in ZCNFG has been discussed. The intention has been to expand on rather than repeat the material in ZCNFG.WS. There you will find rigorous definitions of the data structures used in CFG files. The routines in ZCNFG operate on those data structures. In the next article we'll discuss some of the routines in more detail: how apparently uncalled subroutines are used, indirect calls, menu management, and the use of pointers. Begins to sound like 'C', doesn't it? ●

Reader, from page 52

puters ever since I read the article in *Popular Electronics* on how to build the Altair. Your magazine is full of the same type of people that were doing things in and with computers right after that. Unlike many of the magazines that now publish only IBM PC fluff, you print articles about hardware and software (what computers are all about).

I've just picked up 3 Cromemco S-100 systems and plan to use them for a CAD/CIM in my home workshop, if I ever get the time. By day, I'm a Tool and Die maker and at night build motors and play with computers.

S.B. Phelan CA

Thanks for the compliments, S.B.! Yes, TCJ is filled with the right kind of people, and that includes you. The difference between the TCJ reader and most others is that our folk prefer to roll their own solutions to a problem than to pull out the checkbook for someone else's fix.

Care to tell us more about that project you have lined up for the Cromemco's?—Ed.

———

Hacker's query: How can I read a sector of a Commodore 1581 disk into a buffer using an AT clone's 1.44MB drive? The shareware program ANADISK usually works, but it doesn't do what I want to do. INT 13h doesn't seem to work.

J.R. San Francisco CA

Bill Woodall, our technical advisor for MS-DOS, says that he doubts you can reliably read this disk using the standard AT floppy controller. The COPY-II PC Deluxe Option Board from Central Point Software allows for reading GCR formatted disks such as are used on Commodores, Apple II's and MacIntoshes and comes with the appropriate utilities.—Ed.

———

I have two computers at home: a "stock" Morrow MD-3 and a Kaypro 2X with NZCOM and ZSDOS, Advent TurboROM, 1MB RAM disk, 32MB hard disk and one of those rare MicroSphere Color Graphics boards.

Since we lost *Profiles* and *Micro Cornucopia*, you guys are about all that's left supporting CP/M and the Z-System. Long live the Z-System!

J.W. West Lafayette IN

I and my Ampro Z80's agree with you wholeheartedly.—Ed.

———

The idea of promoting *TCJ* within user groups is very good: the articles lend themselves as very good subjects for discussion during meetings and provide a common background to everyone willing to participate.

L.V.H. Mill Creek, WA

I am a big proponent of user groups but have found that most rely on a few individuals for all their advanced topics. Using a journal such as TCJ can fuel the discussions without overtaxing the group "gurus." I am happy your group is using the journal for just this purpose.—Ed.

See Reader, page 60

Computer Corner, from page 58

program (use a flag in the file header structure to tell the difference). If I went to MINIX directory then, it would forget all the DOS words and load MINIX words. Then I could run MINIX programs, or Forth programs as before.

In either case the MINIX or DOS programs would have to be compiled to run on my hardware of choice, but the actual operating system would be a Forth kernel capable of running the Forth source code directly!

Time Out

I just ran out of space and ideas. Actually I have a few more thoughts on a UOS but am keeping those a closely guarded secret. I'll be waiting to see how much trouble I get into for my open letter. How about you Z80 users, like to see a UOS that you could use as well???●

Computer Corner, from page 64
the operating system. We have seem a new version of OS2 offered up from the DOS camp as a contender for the tittle of *Universal Operating System* as well.

The C people to me have one disadvantage which Forth solves. All the new contenders are based on ports to *new* hardware and will be very large programs. For my Z80 or my 512K 68K system, none of those ports will work. MINIX on a Z80, or a Z180 might be possible but I have my doubts. This is where I feel Forth still has the only chance. Let me explain.

Stack Based Virtual Machine

In Forth the concept is creating a stack based virtual machine that is the same no matter what hardware you are on (a hardware machine done in software). The actual operations performed by the hardware to do the desired operation is of no concern to the programmer. I know that if I put 2 and then 4 on the stack with a plus sign and a period it will print 6 on the screen. I can do that on any hardware port and get the same answer as long as I do not exceed the artificial hardware limits. Artificial I say because the designer or programmer sets the data size based on hardware and standards.

The problem with Forth for some time has been the lack of a specific standard for people to follow.

The problem with Forth for some time has been the lack of a specific standard for people to follow. The problem with any operating system is standards. The problem with the computer industry has been creating standards before a product has died a slow death. In reality all big selling computer systems are not based on standards but what got to market first and also gained the market shares needed to become a standard. MSDOS is an example of a late comer that used a big name and then by default became a standard.

The first product that actually can get some backing and is truly portable to all types of hardware will be the next standard. That means providing for the users something that can not be gotten anywhere else. Remember the first rule of marketing: provide something to the buyer that they can not get elsewhere. A marketing proverb also says the market will not wait for you (he who hesitates is lost).

My Product

It is at this point I give you my *open letter* to Forth system manufactures. It goes like this:

Dear Forth Manufacturer
RE:Universal Operating System (UOS)

I have been reading *Forth Dimensions* lately and have got very tired of seeing people write how they are tired of learning Forth and not being able to use it commercially. The answer I feel is that you manufacturers have let down us users. Many of us went to Forth as a means of bridging hardware systems by doing our programming in what we hoped would become a universal language (non-hardware specific).

I am pleased to see the work going on in getting an ANSI Forth standard. The standard will help but it is taking so long that most Forth programmers will have jumped ship by the time it is finished and adopted by you manufacturers. What we need is for one you who, who already has the products, to produce a Universal Operating System (UOS). Let me explain.

My idea of an UOS would be along the lines of early BASIC and the compiled *run* time module. A number of these *run* time products would be provided by you for each of the current and past hardware platforms. It would be a special version of your current kernel that provides *hooks* for programs to run on (a Forth, BIOS and DOS all combined). It would need a sale price of under $50. To that a vendor writes a program that uses your *hooks* and sells their (encrypted) Forth source code to run on top of it. A typical program could sell for $50 but you must remember it can run on all of the platforms you support.

For a user to write programs for their own use you would sell them a user package, or ANSI Forth that is your normal program riding on top of your run time package. I see a $99 price tag here with some learning Forth type of manuals. The professional programmer buys your $499 package which provides them with all the tools and information needed to write programs for the run time package. The source code gets encrypted and that utility is provided here. Now some users will want embedded applications and as such a separate ROM version of the run time package with a special price of course. Since none of the source has been provided on the internals of the *run* time module, special royalties and pricing would apply for those needing the internal source, probably your current prices you now charge for a metacompiler.

The first product that actually can get some backing and is truly portable to all types of hardware will be the next standard.

With the above pricing and the ability to run programs on any machine directly from the source code, I can actually see a Forth system becoming the path to the Universal language and Operating System.

Thank You,
Bill Kibler

Some Details

The MINIX connection is how I might consider doing the run time module. In UNIX and DOS we have path like structures which are all based on files and their disk location. In Forth we use vocabularies to separate groups of actions or words. I think these two concepts could be one. Why not make your means of searching vocabularies the same as you would search for a file? Let's go even deeper by making a sub-directory contain words that control how you use programs saved as real files.

To explain the last, let's say we have two sub-directories, one called DOS and one called MINIX. If I change to the DOS path, it would load system words that provide the standard DOS interface for programs. I could then run a regular DOS program (if my hardware supported it), or a Forth based

See Computer Corner, page 57

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**

Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**

The renowned Z-System command processor (ZCPR v 3.4) and companion utilities. Dynamically change memory use. Installs automatically
Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

ZMATE **$50**

New Z-System version of renowned PMATE macro editor with split-screen mode for two-window viewing of one or more files. Extremely powerful and versatile macro capability lets you automate repetitive or complex editing tasks, making it the ultimate programmer's editor. Macros can be saved for reuse and also assigned to keys. Editing keys can be reconfigured for personal style. Supports drive/user and named-directory file references. Auto-installs on Z systems. Z-80 only. Supplied with user manual and sample macro files.

PluPerfect Writer **$35**

Powerful text and program editor with EMACS-style features. Edit files up to 200K. Use up to 8 files at one time, with split-screen view. Short, text-oriented commands for fast touch-typing: move and delete by character, word, sentence, paragraph, plus rapid insert/delete/copy and search. Built-in file directory, disk change, space on disk. New release of our original upgrade to Perfect Writer 1.20, now for all Z80 computers. On-disk documentation only.

ZSDOS .... **$75, for ZRDOS users just $60**

State-of-the-art operating system. Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk .... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**

Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**

Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402
(213)-393-6105 (eves.)

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **PluPerfect Writer** ©, **JetFind** © Copyright 1986-88 by Bridger Mitchell.

Reader, from page 57

I found the articles on *Getting Started in Assembly Language* very interesting. I guess it is about time I learned to use it.

J.F. Lake Zurich IL

Al Hawley seems to have hit on a winning topic here. It seems there are a number of people who having been doing programming in high level languages for years that have wanted a bridge to assembler. As the author of the ZMAC assembler, Al certainly knows his subject!—Ed.

——

In response to our brief chat on GEnie tonight, I am enclosing a check for a two year subscription. The article discussing the parallel/SCSI conversion is of particular interest to me.

It certainly proved to be an enlightening 20 minutes or so.

D.E.F. Minneapolis MN

This letter refers to a real-time conference on GEnie in April where I was the guest speaker. I was asked back again in June for a repeat. I certainly enjoyed both conferences and it seems others did as well. Let me encourage readers to join in the CP/M and Forth SIGs on that service.

By the way, I am expecting another article on adding SCSI to computers that lack it, but this time using a wire-wrapped daughter board under the CPU.—Ed.

——

I am using ZSDOS quite a bit and find it to be a great enhancement for CP/M. I note in Jay's column for issue 51 a list of three publications leaning to CP/M and Z-System. I would like to add two more. They are not quite as technical but more oriented toward the guy who is trying to make his old machine do his bidding. These are *The SEBHC Journal* by the Society of Eight-bit Heath Computerists and *The Staunch 89er*. Both of these publications mention *TCJ* as the ultimate for technical information, but I have not seen them referred to in *TCJ*.

S.R.E. Hemet, CA

Thanks for mentioning these two publications. You are right, they are very good. I subscribe to one of them myself, and should send an order in for the other. Though their titles talk to Heath computers, they are good general-information CP/M rags. I recommend them both. The address for SEBHC Journal *is 895 Starwick Drive, Ann Arbor MI 48105.* The Staunch 8/89er *has an advertisement in the Marketplace page of this issue.—Ed.*

——

Your comment about the one company that sells postage meters caused me to write this letter. It just ain't so. Since we all know about Pitney-Bowes, I must assume you think they are the only postage meter company around. There is also Friden, and two others.

B.C. Chicago IL

We learn something new every day. Thanks.—Ed.

Editor, from page 2

I was a bit embarrassed when I published the Z-Node listing back in issue 50. Bob Dean's system was shown as down. Even though I had called and gotten through the day before we went to press, I forgot to correct his entry. So when Ian Cottrell was asked by Jay Sage to release the next edition, I volunteered to verify the US systems.

There is a system on the US west coast that is shown in the RCPM listing as being a Z-Node, but which is not in the Z-Node list itself. I logged on, explained what we were doing and asked if I should show the system or not. The answer surprised me.

No, the sysop said, he had dropped his affiliation to protest the commercialization of Z-System since the demise of Echelon.

What? What commericialization? Let's look this over.

Echelon was a hold-out of commercial support for Z-System before closing its doors back in 1987. Though they allowed for free distribution of ZCPR source code, they charged for everything else. It was a company like any other, and needed to make a profit to survive. And while their policy of giving the source code away was a noble gesture, it also contributed to the unprofitability of the company to a point where it did not survive.

Question: As Echelon charged for their products with the exception of ZCPR source, were they not commercializing Z-System?

Question: Did it serve the public good to have the company fail?

Since that time, a few enterprising people have picked up the ball and continue to support Z-System. NZCOM, which replaces Z-COM (always a commercial product, by the way), ZSDOS, BackGrounder II are examples of their products. Would it be in the public interest that these products not be available?

Keep in mind, while CP/M has a heritage of a strong public domain, this extends to utilities and a few applications. With the exception of the original ZCPR source code, operating systems have never been public domain. The same is true of languages and assemblers. Do you believe Bill Gates gave away M80 and L80? Was Gary Kildall giving away CP/M 2.2? Have you called Digital Research lately to get a copy of PL/I? They'll ask for your $500 check first, as will Ashton Tate for dBase-II.

A related string of messages have been floating down the wire on the Internet comp.lang.forth newsgroup. A fellow was looking for an alternate source for F68HC11 CPUs. It seems he didn't care to pay $25 more for an F68HC11 than for a straight 68HC11. Well, let's look at this. The F68HC11 has Forth built into it and is proprietary to New Micros. Essentially, you are paying $25 to New Micros for their code. Should we expect New Micros to work for free? Would you quit your job, work for free and pay the bills necessary to support your users? My only wonder is why New Micros gives you such a powerful chip so cheaply.

Where is this tirade leading? When we cry for lack of support yet damn those who will stand by us, we are hypocrites. If we begrudge a company sufficient income to pay its bills, we are calling for its failure. Most of the companies that support our systems, whether they be Forth or Z-System, are very small. Not many are getting rich at this.

We come then to the opening question: When is the Public Domain not in the Public Interest? The answer: when attitudes are created that require all products be provided for free, thereby forcing the failure of any company that attempts to support the public.

Thank the public domain programmer. This person is truly a treasure. But do not condemn someone for needing to support a family when you are needing his labor. Not all can work for free.

The CPU280

A High-Performance Single-Board Computer

By Tilmann Reh

The CPU280 is a complete computer based on Zilog's high-performance Z280 CPU on a single 160 x 100 mm board (EuroCard). The circuit is designed to give maximum performance without unnecessary circuit expense. Since the Z280 is fully software compatible with the Z80, all Z80 application software can be run without modification. New software can be developed to use the much more powerful instruction set of the Z280, further increasing the computer's performance.

The CPU280 consists of a PCB with the CPU itself, all necessary memory, two serial interfaces, a real-time clock with non-volatile RAM, a floppy controller, and a bus interface. The circuit design is very straightforward with fully synchronous timing signals, which results in good operating stability. Since all parts are CMOS, the power requirement is very low: running at maximum clock speed, the whole board draws about 350 mA from a single 5 V supply.

The CPU is run in 16-bit Z-bus mode, the fastest available bus option for the Z280. Clock frequency is up to 12.288 MHz, with the on-board memory bus running at full clock speed. There are two EPROM's with 64 or 128 kB capacity on board, providing boot/system software. Dynamic RAM may be configured as 512k, 1M, 2M, or 4M bytes, depending on the type and number of RAM chips used. All memory is 16-bit wide, and the dynamic RAM supports the processor's burst mode (reading 4 words of instructions with two memory cycles).

The internal serial interface of the Z280 is completed with external handshake signals and line drivers to build a complete RS-232C interface with hardware handshaking. Another serial interface chip on the board adds a second RS-232C interface. Both serial interfaces are fully programmable in data format and data rate (up to 38400 bps each), and both are capable of interrupt-driven operation.

The real-time clock (RTC) supports date and time information required by modern operating systems. It can also provide periodic interrupts or an alarm interrupt at a preprogrammed date and time. A lithium battery maintains power to the RTC and to 50 bytes of RAM. This NV-RAM is ideal for storing setup and configuration parameters.

The on-board floppy disk controller (FDC) is able to control up to four drives of any type and in any combination. The maximum data rate is 500 kbps, which results in maximum disk capacities of 1.44 MB (5.25 inch) and 1.76 MB (3.5 inch) when using high-density drives. Double-density disks in 5.25-inch drives with fixed rotational speed of 360 rpm are also supported (data rate 300 kbps). Floppy I/O can be handled with one of the four DMA channels of the Z280.

External I/O extensions are supported with an ECB-Bus interface. As the address decoding uses an extra I/O address page for the bus interface, all 256 I/O addresses of standard ECB are available. The bus clock is half the memory clock (6.144 MHz max.), with timing signals stretched to meet the requirements of standard Z80-B peripherals. External interrupts are supported, in any of the Z280's interrupt modes. Further on-board I/O includes three LED's functioning as status indicators and three software- readable jumpers (i.e., for configuration purposes).

There is a complete CP/M-3 (CP/M-Plus) installation available, using every feature of the board. Besides the CPU280 board and this software, you need only a disk drive and a CRT terminal to build a complete high-performance CP/M-3 workstation (performance is comparable to a Z80 running at 15-20 MHz, depending on software). The cold-boot loader supports various configurations (using the NV-RAM) via it's setup menu. Normally, the whole operating system, including the command processor (CCP), is booted straight from the EPROM; for upgrading or testing, disk boot is also possible. Both the boot loader and the operating system support flexible disk format changes using the 'AutoFormat' feature of the BIOS. You are able to swap freely between many different formats. The menu-driven 'FormatManager' program handles format definitions and disk formatting and checking.●

Who Do You Love?

I have a custom of giving a word of thanks to those whose help has been instrumental in putting *TCJ* out. This month, there are three people I want to shine the spotlight on. Myla, Jennifer and Cathy. These three young ladies are, if you will, our shipping department. Myla acts as supervisor while the other two struggle frantically to get all the issues addressed and ready for the post office. What had been taking me three days took them one evening. Should I mention that the three are my daughters? No man could ask for finer children.

Thanks, kids!Pizza's on me.●

Real programmers don't document.
If it was hard to write,
it should be
hard to understand.
—Anonymous

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 8:
- 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.

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.

The Computer Journal

Back Issues

Sales limited to supplies in stock.

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 Multi-tasking & 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

Issue Number 51:
- Introducing the YASBEC
- Floppy Disk Alignment w/RTXEB, Pt 3
- High Speed Modems on Eight Bit Systems
- A Z8 Talker and Host
- Local Area Networks—Ethernet
- UNIX Connectivity on the Cheap
- PC Hard Disk Partition Table
- A Short Introduction to Forth
- Stepped Inference as a Technique for Intelligent Real-Time Embedded Control
- Real Computing, the 32CG160, Swordfish, DOS Command Processor
- PMATE/ZMATE Macros
- Z-System Corner, The Trenton Festival
- Z-Best Software, the Z3HELP System
- 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

The Computer Corner

By Bill Kibler

I have a few topics to cover and an open letter to explain. It has been rather busy lately and I have been doing some thinking about my choices in operating systems. That thinking usually gets me into trouble, but then life would be boring without a little trouble.

68KEForth

I have put a running version of 68K assembly based EForth on GEnie and it has been down loaded over ten times so far. I did not find any messages to me about it (or anything for that fact...) which makes me feel that no one has found anything wrong yet.

It was interesting to note how it worked ok on a 68020 but not a 68000. All the problems were the UM/MOD or unsigned divide function. Everything uses this function and so little worked until I changed it. The difference between the processors was in aligning the code. The 68000 must have longs on even boundaries, while a new feature of the 020 does not care about even boundaries.

Chuck Moore said how it made all the difference when he separated the return stack from the data stack.

It never seems to amaze me that I always learn something new (or forgotten from the past) when I do these ports. I had studied the 020 but had not really understood the difference between it and the 68000 until I tracked down the problem. We are trying to find an interrupt based bug in our 68K code at work and again I thought I understood interrupts. Well I really understand them now, and can even follow an operation back many levels. Normally, this would not be a problem, but our code does some really bad stack games when they push parameters.

Let me say here and now that the best way to pass data on stacks is to have two separate stacks. Chuck Moore (creator, if you will, of Forth) said how it made all the difference when he separated the return stack from the data stack. I would say that for most programmers, their programs are seldom so complex that separate stacks would mean little to them. Our programs are so multi-threaded and complex that it is amazing that it runs using one stack.

Actually, there are many stacks in the program (one for each process running) but all data and return addresses are on them. A most bazaar method of passing pointer to pointers on the stack has us adding to return addresses, calculating displacement of saved register data, offsetting around storage pointers, tracking the number of items passed, trapping if too few or too many items passed, and more. Before using Forth and now after using it for years, I always felt that data should be on a separate stack from the system or return stack. Working on this program has shown just how right a decision that was.

Well here comes my regular lecture about *macros* and how bad they are. I think it was a contributing factor in using such a bad stack method. Some person wrote a macro to do all this fancy stacking and un-stacking and since no one could figure it out they were all forced to use it. Lets see, there are 200 files which probably use this macro 3 to 4 times in each file and typically there are 4 separate process stacks, with over $40 (64) bytes for each time it is used. That makes these stack very long and probably slows the overall operation down by 30%. A simple push of the needed items, even if blank values, would be many time faster.

Universal DOS?

Well I have started doing my MINIX port onto a SAGE/STRIDE computer. First, let me say if anyone has done it already please drop me a line so I can stop. As I start moving on the project, I can see how it will be a learning experience only. The newer version (1.5) needs more memory than my 512K 68K machine and will have to stay at the old level (1.3). What I need is a smaller and more compact DOS. MINIX has some good ideas (well really UNIX) and that got me thinking (trouble time again).

What got me into Forth and now MINIX was trying to set up a universal operating system. I have so many different systems that I wanted a single DOS that would allow me to use a single program (and disk format) on any of those systems. Forth offers that option, but most ports have no operating system or consist of other constraints preventing commonalty between systems. MINIX might give you the commonalty of operations between systems, but all programs would have to be recompiled (the C programs that is) before working on a different port. I have also seen in both systems problems caused by hardware variations making each port difficult with lots of special attention to make it work.

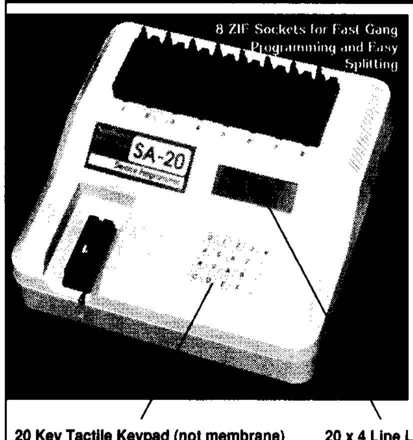
POSIX and other groups are trying to make just what I want to some degree.

If you have been following any of the computer magazines you will know that my wants are actually pretty well universal. POSIX and other groups are trying to make just what I want to some degree. The groups are basing it on C as their language which does have some merits and UNIX as

See Computer Corner, page 58

EPROM PROGRAMMERS

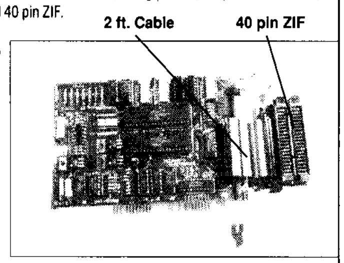
Stand-Alone Gang Programmer $750.00


8 ZIF Sockets for Fast Gang Programming and Easy Splitting
SA-20
20 Key Tactile Keypad (not membrane)
20 x 4 Line LCD Display

- 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	Hardware Interfacing
Operating Systems	Proprietary Languages
Image Processing	Component Lists

Custom Software Solutions for Business:

Order Entry	Point-of-Sale
Warehouse Automation	Accounting Systems
Inventory Control	Local Area Networks
Wide Area Networks	Telecommunications

Publishing Services:

Desktop Systems	Format Conversions
Books	Directories
CBT	Interactive Video

33 North Doughty Ave, Somerville, NJ 08876 • (908) 526-5980

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- Automatic, Dynamic, Universal Z-Systems: Z3PLUS for CP/M-Plus computers, NZCOM for CP/M-2.2 computers ($70 each)
- XBIOS: the banked-BIOS Z-System for SB180 computers at a new, lower price ($50)
- PCED — the closest thing to Z-System ARUNZ, and LSH under MS-DOS ($50)
- DSD: Dynamic Screen Debugger, the fabulous full-screen debugger and simulator, at an incredible new price, down from $130 ($50)
- ZSUS: Z-System Software Update Service, public-domain software distribution service (write for a flyer with full information)
- Plu*Perfect Systems
  - Backgrounder ii: CP/M-2.2 multitasker ($75)
  - ZSDOS/ZDDOS: date-stamping DOS ($75, $60 for ZRDOS owners, $10 for Programmer's Manual)
  - DosDisk: MS-DOS disk-format emulator, supports subdirectories and date stamps ($30 standard, $35 XBIOS BSX, $45 kit)
  - JetFind: super fast, extemely flexible regular-expression text file scanner ($50)
- ZMATE: macro text editor and customizable wordprocessor ($50)
- BDS C — including special Z-System version ($90)
- Turbo Pascal — with new loose-leaf manual ($60)
- ZMAC — Al Hawley's Z-System macro assembler with linker and librarian ($50 with documentation on disk, $70 with printed manual)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Z80 assemblers using Zilog (Z80ASM), Hitachi (SLR180), or Intel (SLRMAC) mnemonics, and general-purpose linker SLRNK
  - TPA-based ($50 *each* tool) or virtual-memory ($160 *each* tool)
- NightOwl (advanced telecommunications, CP/M and MS-DOS versions)
  - MEX-Plus: automated modem operation with scripts ($60)
  - MEX-Pack: remote operation, terminal emulation ($100)

Next-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $3 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (pw=DDT) (MABOS on PC-Pursuit)