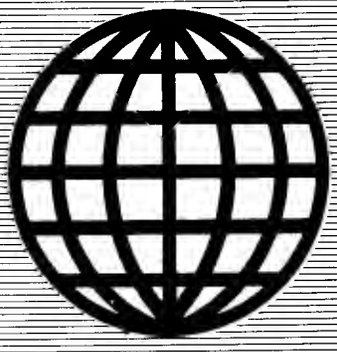


The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 34 September / October 1988 $3.00

Developing a File Encryption System
Keep MS-DOS Files Private with a Customized Encoding Scheme

Data Base
Choosing Your Tools

A Simple Multitasking Executive
Designing an Embedded Controller Multitasking System

CP/M's Magical IOBYTE
Using Tables for Space Efficient Implementation

ZCPR3 Corner
PRL Files and Type-4 Programs

New Microcontrollers Have Smarts
Program with ROM Based On-Chip BASIC or Forth

Advanced CP/M
Extending the Operating System

Data File Conversion
Converting Macintosh Files with Turbo Pascal

ISSN # 0748-9331

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, Montana
59912
406-257-9119

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Production Assistant
Judie Overbeek

Contributing Editors
Joe Bartel
Bob Blum
Bill Kibler
Rick Lehrbaum
Bridger Mitchell
Jay Sage

The Lillipute Z-Node sysop has made his BBS systems available to the TCJ subscribers. Log in on both systems (312-649-1730 & 312-664-1730), and leave a message for SYSOP requesting TCJ access.

***Subscription rates**—$16 one year (6 issues), or $28 two years (12 issues) in the U.S., $22 one year in Canada and Mexico, and $24 (surface) for one year in other countries. All funds must be in US dollars on a US bank.*

Send subscriptions, renewals, or address changes to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, Montana, 59912, or The Computer Journal, PO Box 1697, Kalispell, MT 59903.

Address all editorial and advertising inquiries to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912 phone (406) 257-9119.

The COMPUTER JOURNAL

Features

Issue Number 34
September / October 1988

Columns

Editor's Page

Industry Trends

The microcomputer market is dominated by the IBM PC-DOS and Apple Macintosh systems, with Atari and Commodore playing a secondary low-level role. The majority of the consultants and custom programmers I talk to are working with PC/MS-DOS systems. Some of them are using Macs, but very few are programming for them.

When you are in the business of consulting and developing programs for other people's computers you have to work with the systems which they have chosen, and not what you would choose for yourself. A study of the computer publications shows that the IBM-PC related publications cover both end user products and programming tools and techniques, while the Mac publications concentrate on the end user products. Apparently, the Macs are selected by people whose needs are filled by existing software such as spreadsheets, wordprocessors, and desktop publishing, while the people who need custom programming use the PC.

The majority of the Mac and PC programs have a fundamental difference. Most Mac programs stress the ease of use, and because they avoid any confusing complications in the form of user decisions, they can only do the tasks in the manner that the designer implemented them. While there are also some simple-minded PC programs, many PC programs require more involvement by the user, and can handle more complicated tasks. Because it takes a lot of time and effort to develop a program for the Mac, most of the Mac development is being done by larger organizations. Individual innovative entrepreneurs do not have the resources to develop Mac applications. Or perhaps, innovative individual programmers do not like the Mac atmosphere. PCs, on the other hand, are being programmed by many individuals, and there is a much wider range of programs available. There are exceptions, of course, but in general terms, the Mac is for users, and the PC is for both users and programmers.

The relative status of the Mac and the PC is changing, as more people become familiar with the Mac programming environment and more Mac programming tools are sold. There are also increasing numbers of PCs being purchased for use in applications by people who have no intention of programming. As the trend towards Mac-like interfaces for the PC continues, and as OS/2 or other '386 operating systems with multitasking, windows and icons become more common, the PC will become as difficult as the Mac to program.

"The driving force in the mass markets is not the features of the system, but rather the software which is available."

It will be interesting to see what happens with OS/2. Most developers do not like it, but will be forced to use it if the market chooses it. There are better systems available, but using them is pretty much a custom programming situation.

There is no apparent successor to the Mac and the PC at this time, and we will probably be using these systems for the next two to five years. At which time they will have such a large established base, that we will most likely see improvements to these basic systems instead of the major changes we encountered when switching from Apple II to Mac or CP/M to PC.

The driving force in the mass markets is not the features of the system, but rather the software which is available. The developers follow their perception of their market, and the market (generally) follows the availability of the software. Hardware has to be sold to develop a market for the software, and people won't buy the hardware until the software they need is available. These facts make it extremely difficult for someone without the marketing power and resources of Apple or IBM to break into the market with a new system.

We're Going Desktop

I've been fighting our old archaic Compugraphic typesetter, waiting till the DTP (Desk Top Publishing) hardware and software evolved to the point at which I would be comfortable with it. The time has finally come, and we will be changing over to DTP.

The first DTP programs performed a lot of amazing tricks, but they were designed for producing letters, reports, and resumes, and they did not provide the typographical refinements I wanted. Now, the current revisions of the programs have eliminated most of the objectionable faults of the original releases. I still held back, hesitant to give up something with which I was familiar. The final push came when the local paper installed a system which can output to either a laser writer or a phototypesetter. Modern technology finally arrived in Northwest Montana!

Going to DTP obviously required a new computer system. I would have to get something besides a CP/M system (I now have seven of them!). The big decision was whether to go with an MS-DOS or Mac machine. I had been thinking about this for some time, and had reached the decision that I was more comfortable with the MS-DOS machines. A major factor in the decision was that only a small portion of the time would be spent using the actual DTP programs. Most of the time would be spent in wordprocessing, and I am very fond of WordStar. If you think that Revision 4 was an improvement, wait till you see Revision 5 (sorry, MS-DOS only, not CP/M). It is so good, that I was not willing to give it up by going to a Mac system—I'll have more on WordStar Revision 5 after I wring it out. MicroPro has been very busy, and has given me a lot to work on. Another very important factor was that the primary non-editorial use will be for C and data base programming, and there are a lot more tools available from multiple sources for the PC than there are for the Mac.

We won't be purchasing an output device for a while. It will be a nuisance to run out for printouts—especially since everyone in this remote area interfaces to a Mac. I may have to transfer the files to a

Mac disk in order to get output. A modem? No such luck. The nearest place with telecommunications is almost 400 miles away, and I don't want to face the problems and delays which that would involve. I would rather transfer the files and drive 15 miles to where I can deal with things one-on-one.

Getting into laser printers is a whole new ball game with lots of new buzz words and jargon. Postscript® vs. PDL. White printing vs. black printing. Lots of new things to consider. There are a lot of laser printers being sold at very attractive prices—much less expensive than the Apple LaserWriter® . But be careful, and be sure you know what you are buying! All of the cheap ones which I have seen use PDL instead of Postscript, and everyone is changing over to Postscript. It is something like the non-IBM compatible MS-DOS machines sold a few years ago (and still being dumped very cheap). If you get a PDL machine you'll end up with an orphan, so don't get anything except a Postscript machine unless you really know exactly what you are getting into.

A New Computer

I had decided that the new system should be MS-DOS, but that still left a lot of choices. The first decison was to select the CPU. 8088, 80286, or 80386? I chose an AT clone with the 80286 because the 8088 systems are too slow for the DTP programs which make heavy use of graphics and require a high resolution monitor, and the '386 systems are too expensive. Another criterion was that I wanted something which I could subject to heavy use without any problems. There have been some real lemons on the market, and I wanted to avoid them.

I selected the Austin '286 (contact Dwayne Derrick, Austin Computer Systems, 7801 N. Lamar, Suite D-95, Austin, TX 78752, 1-800-752-1577). It runs at 12.5 Mhz, comes with 1 Meg on board expandable to 4 Meg on the board, has two serial ports, parallel port, PS/2 mouse port, hard and floppy disk controllers, and the graphics, all on the mother board. It uses the Western Digital six layer motherboard which is currently made here in the U.S. The wafers are made here, packaged overseas, and returned here for stuffing using surface mount technology.

Austin supplied a copy of the Western Digital Tech Manual (120 pages) with good solid information. The parallel port is bidirectional, so it can be used for both input and output, the hard drive controller uses 1:1 interleaving, and there is a 32 Kbyte disk cache.

I chose this product for three reasons: 1) It is a high quality product with advanced technical features, and it is cost competitive. 2) I'll support U.S. products where they are available. 3) I talked to people at both Austin and Western Digital who had good product knowledge, and were courteous and interested in solving any problems (there weren't any problems, I just called to check their response).

The third point was very important to me. I called some other vendors with uniform poor results.

If you're upgrading to an AT, want something that works instead of a constant hardware troublshooting project with questionable compatibility, and value support, give Austin a call.

Training Classes Help

I've learned most languages and programs by spending long hours at the keyboard with the books and manuals. One of the reasons that I've used this method is because of our isolated geographic loctaion. The other reason is that too many of the training sessions are so poorly done that they are useless—the audience knows more about the product than the trainer.

I recently started learning Raima's db_Vista III, and took advantage of the opportunity to attend their one week class in Bellevue Washington. It was very worthwhile, and was a good example of what a class should be. The Monday and Tuesday basics class is usually given by Mark Hervol, but he was not feeling well, so we had Randy Merilatt who is the president and co-developer of the product. Mark was feeling better and gave the Wednesday advanced class. Wayne Warren, vice president of development and the other co-developer, gave the Thursday and Friday internals class.

Needless to say, Randy and Wayne really knew the product, since they developed it. Mark also had good product knowledge, because he actually writes applications using db_Vista. Frequently, the people who developed the product and wrote the code can no longer relate to the users, but all three of the trainers were very good at listening to the class and in understanding the questions. When there was a question which they could not answer, they didn't try to bluff their way through. They either dug the information out of the manuals, or said that they would have to get the information—and then they did get the information back to us!

db_Vista is a very powerful network model DBMS which is used in C programs, and the source code is available from Raima. Taking the class will make it much easier to use than if I had tried to learn it from just the manuals.

The company, its people, and their attitude are very important factors to consider when selecting a product. I talked to a lot of Raima people besides Randy, Wayne, and Mark. I also talked to users who had dealt with both the product and the company. I left with good feelings about Raima, and feel that they are really interested in working with their customers.

Data-on-a-Disk

The high capacity CD ROM optical disks are attracting a lot of attention, and everybody is talking about the amount of reference material that can be packed on one disk. But, there is another development which I have not seen mentioned anywhere.

A number of companies are starting to market specialized reference material on floppy disks. While the CD ROM can store huge amounts of data, there are a lot more floppy drives than there are optical drives, and the high capacity floppies can hold the needed data for a narrowly focused special interest.

A few of the examples which I remember are: 1) A data base of short run printers. We purchased this for use in locating printers for our books. 2) Business Week Top 1000 companies listing the top officials and financial data. 3) Tide data for the U.S. Coasts. 4) Nutritional database for 4,589 foods. 5) Rare stamp information and valuations for collectors.

I have been considering this market for some time, and feel that it has real potential for a small publisher. The largest markets will be appliance computer users who are interested in following their interests, but don't want to become computer or database experts. In order to serve this market, we'll have to provide a complete ready-to-run product which includes the data plus the database management system—this is one reason why I am concentrating on DBMSs which do not require a run time licensing fee. For now, I'll be concentrating on PC applications, but I'll be looking back over my shoulder at Hypercard® on the Mac because it may really take the lead in this field.

We would be very interested in articles on these types of databases, or on Hypercard concepts and design. ■

Developing a File Encryption System

Keep MS-DOS Files Private with a Customized Encoding Scheme

by Dr. Edwin Thall

Dr. Edwin Thall, Professor of Chemistry at The Wayne General and Technical College of The University of Akron, teaches chemistry and computer programming.

Some excellent commercial packages for encrypting data files are available. Not only do they scramble data, but many offer a variety of options including unique passwords for different users, passwords that don't work after a certain time frame, a log indicating which files were accessed and by whom, directories which cannot be copied, and inoperable drives. Typically, these utilities modify the operating system and require special installation.

Many users may not need an elaborate computer security system, but rather some reasonable method to keep out the casual snoop. A customized encryption system may be the answer, especially if you do not store state secrets, and enjoy the challenge of programming. In this article, I explain how to scramble data, as well as how to develop your own customized MS-DOS encryption/password system. Also, ENCRYPT.EXE is presented, a utility that should satisfy the needs of many users.

Encryption By Rotation

Although it is relatively easy to encrypt data, keep in mind that scrambled data is useless to everyone, including the rightful owner. Therefore, any potential file encryption scheme must be entirely reversible. Two operations, rotate and XOR, are well-suited for this game.

Let's begin by rotating bits, first to the right and then to the left. The ASCII code for character "A" is 41H in hexadecimal and 01000001B in binary. If contained in the AL register, these eight bits rotate two places to the right when the following instructions are invoked:

```
MOV   CL,2
ROR   AL,CL
```

Successive rotations modify the contents of the AL register from 41H to 50H in two steps:

```
0100 0001  ----->  1010 0000  ----->  0101 0000
  41H                AOH                50H
  ''A''              '' ''              ''P''
```

The original data (41H) is readily restored when bits in the AL register are rotated two places, but, this time, to the left.

```
MOV   CL,2
ROL   AL,CL
```

A short program, written in assembly language, to scramble eight consecutive bytes of data is provided in Figure 1. From DEBUG, use the assemble command to enter the program (omit comments):

```
A>DEBUG
-A100
```

After typing in the code, display the eight bytes of data (offsets 0115-011CH), run the program, and then display the data again:

```
-D115,11C
DS:0115   53 43 52-41 4D 42 4C 45        SCRAMBLE
-G
-D115,11C
DS:0115   D4 D0 94-50 53 90 13 51        TP.PS..Q
```

```
DS:0100  MOV   BX,0115       ;point to data
DS:0103  MOV   CX,0008       ;count 8 bytes
DS:0106  PUSH  CX            ;save count
DS:0107  MOV   AL,[BX]       ;move data to AL register
DS:0109  MOV   CL,2          ;rotate AL register
DS:010B  ROR   AL,CL         ;2 times to right
DS:010D  MOV   [BX],AL       ;restore coded data
DS:010F  INC   BX            ;point to next byte
DS:0110  POP   CX            ;restore count
DS:0111  LOOP  0106          ;process next byte
DS:0113  INT   20            ;return
DS:0115  DB    ''SCRAMBLE''    ;data
```

Figure 1. Assembler code to rotate data two bits to the right.

Notice how every byte was rotated two places to the right and the original data (SCRAMBLE) now exhibits gibberish (TP.PS..Q). To restore the original data, substitute ROL for ROR, execute the program a second time, and display the data:

```
-A10B
DS:010B  ROL  AL,CL
<ENTER>  2X
-G
-D115,11C
DS:0115   53 43 52-41 4D 42 4C 45        SCRAMBLE
```

As you can see, an equivalent ROL cancels a previous ROR, and vice versa. Since bits can be rotated right or left, from 1 to 7 times, the following parameters are possible:

```
1R  2R  3R  4R  5R  6R  7R
1L  2L  3L  4L  5L  6L  7L
```

Rotating the bits eight places in either direction (8R or 8L) results in no change in the bit pattern.

To increase the number of encryption possibilities in the previous program, you could specify different rotation

parameters to sequences of data. For example, rotating the odd bytes twice to the right and even bytes five times to the left (2R5L) represents one of 256 possibilities (16^2). But keep in mind that four of these combinations (8R8R, 8R8L, 8L8R, or 8L8L) give no change in the data.

ROT2X.COM, listed in Figure 2, demonstrates how to individually rotate odd and even bytes. The data, originally 10 consecutive bytes of 41H, is scrambled by specifying the parameters. Remember to convert ROT2X.ASM into ROT2X.COM and to enter R/L in caps. Each time new parameters are input, the modified data is displayed on the screen. To terminate the program, you must hit Ctrl break. Try running the program with parameters 2R5L:

```
A>ROT2X
ENTER PARAMETERS (2R5L, 8R3R, etc.):  2R5L
P(P(P(P(P(
ENTER PARAMETERS (2R5L, 8R3R, etc.):  ^C
A>
```

The five odd data bytes were changed from 41H ("A") to 50H ("P"), while the five even data bytes altered from 41H ("A") to 28H ("("). To restore the original data, the directions are reversed with the odd bytes rotated twice to the left and even bytes five places to the right (2L5R):

```
A>ROT2X
ENTER PARAMETERS (2R5L, 8R3R, etc.):  2L5R
AAAAAAAAAA
ENTER PARAMETERS (2R5L, 8R3R, etc.):  ^C A>
```

If the wrong parameters are entered during the restoration procedure, the data is rotated a second time and rendered virtually useless.

ROT2X can be easily modified to ROT3X, ROT4X, ROT5X, etc. For example, ROT5X requires the input of five parameters in the form 2R3L4R6L7L. With over one million (16^5) combinations possible, guessing the algorithm—the special formula to solve the problem—is unlikely.

Encryption By XOR

XOR stands for "exclusive or" and refers to either one or the other, but not both. In terms of how it operates on bits, XOR behaves like this:

```
0  XOR  0  =  0
0  XOR  1  =  1
1  XOR  0  =  1
1  XOR  1  =  0
```

When 41H is XORed with 03H, 42H is returned:

```
0100 0001   41H    ''A''
0000 0011   03H    XOR operand
---------
0100 0010   42H    ''B''
```

If 42H is XORed again with 03H, the original value (41H) is restored:

```
0100 0010   42H   ''B''
0000 0011   03H   XOR operand
---------
0100 0001   41H   ''A''
```

Because consecutive operations scramble and unscramble data, the XOR operation is ideal for file encryption. In the previous examples, both data and code were stored in the same program. However, an effective utility must read data from an external source, scramble the data, and then write the encoded data back to the original file. The next example, utilizing separate programs for code/data, demonstrates encryption by means of the XOR instruction.

First, let's create a simple data file. From DEBUG, generate 10 bytes of 41H, write the file to disk as A:TEST, and display its contents:

```
A>DEBUG
-NA:TEST
-F100,109 41
-RCX
CX ????
:00A0
-RBX
BX ????
:0000
```

```
;ROT2X.ASM converts to ROT2X.COM
;Select rotation parameters for odd/even bytes.
;Hit Ctrl Break to terminate program.
CSEG    SEGMENT
ASSUME CS:CSEG,DS:CSEG,ES:CSEG
        ORG    100H
START:  JMP    SKIP              ;skip data
MESS    DB     0AH,0DH,'ENTER PARAMETERS (2R5L,8R3R,etc.):   $'
PARM    DB     5,5 DUP(''P'')
PRINT   DB     0DH,0AH
DATA    DB     10 DUP(''A''),0DH,0AH,24H
;Display message to enter rotation parameters
SKIP:   MOV    AH,9
        MOV    DX,OFFSET MESS
        INT    21H
;Input parameters
        MOV    AH,10
        MOV    DX,OFFSET PARM
        INT    21H
;Scramble data
        MOV    BX,OFFSET DATA    ;point to data
        MOV    CX,5              ;count 5 passes
ROT:    PUSH   CX                ;save count
        MOV    AL,[BX]           ;odd byte to AL
        MOV    CL,PARM+2         ;get odd rotation value
        SUB    CL,30H            ;convert ascii to hex
        CMP    PARM+3,''R''        ;get odd direction
        JZ     RIGHT             ;jump if direction right
        ROL    AL,CL             ;rotate left
        JMP    NEXT              ;jump if direction left
RIGHT:  ROR    AL,CL             ;rotate right
NEXT:   MOV    [BX],AL           ;restore odd byte
        INC    BX                ;point to even
        MOV    AL,[BX]           ;even byte to AL
        MOV    CL,PARM+4         ;get even rotation value
        SUB    CL,30H            ;convert asciiz to hex
        CMP    PARM+5,''R''        ;get even direction
        JZ     RITE              ;jump if direction right
        ROL    AL,CL             ;rotate left
        JMP    NEXT2             ;jump if direction left
RITE:   ROR    AL,CL             ;rotate right
NEXT2:  MOV    [BX],AL           ;restore even byte
        INC    BX                ;point to odd byte
        POP    CX                ;restore count
        LOOP   ROT               ;process next 2 bytes
;Display data
        MOV    AH,9
        MOV    DX,OFFSET PRINT
        INT    21H
        JMP    SKIP              ;repeat entire process
CSEG    ENDS
END     START
```

Figure 2. Assembler code for ROT2X.COM.

```
;CODE.ASM converts to CODE.COM
;This program XORs data in A:\TEST
;Select XOR hex code (01-FFH) in CAPS
CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
        ORG     100H
START:  JMP     SKIP                ;skip data
MESS    DB      0DH,0AH,'SELECT XOR CODE (01-FF):   $'
CODE    DB      0
ASCIIZ  DB      'A:\TEST',0
HANDLE  DW      0
PRINT   DB      0DH,0AH
BUFFER  DB      10 DUP (0),24H
SKIP:
;Display message for XOR hex code
AGAIN:  MOV     DX,OFFSET MESS
        MOV     AH,9
        INT     21H
;Input and store XOR hex code
        MOV     AH,1                ;input high digit
        INT     21H
        CMP     AL,30H              ;check for hex
        JB      AGAIN               ;if wrong, try again
        MOV     DL,AL               ;store high in DL
        SUB     DL,30H              ;change ascii to hex
        CMP     DL,9                ;compare with 9
        JBE     DIGIT1              ;if less/equal, skip
        SUB     DL,7                ;subtract 7
        CMP     DL,9                ;check for hex
        JBE     AGAIN               ;if wrong, try again
        CMP     DL,16               ;check for hex
        JAE     AGAIN               ;if wrong try again
DIGIT1: MOV     CL,4
        SHL     DL,CL               ;shift DL 4 bits to left
        MOV     AH,1                ;input low digit
        INT     21H
        CMP     AL,30H              ;check for hex
        JB      AGAIN               ;if wrong, try again
        SUB     AL,30H              ;change ascii to hex
        CMP     AL,9                ;compare with 9
        JBE     DIGIT2              ;if less/equal, skip
        SUB     AL,7                ;subtract 7
        CMP     AL,9                ;check for hex
        JBE     AGAIN               ;if wrong, try again
        CMP     AL,16               ;check for hex
        JAE     AGAIN               ;if wrong, try again
DIGIT2: ADD     AL,DL               ;add high/low
        MOV     CODE,AL             ;store code
;Open A:\TEST
        MOV     AH,3DH              ;open function
        MOV     AL,2                ;read/write access
        MOV     DX,OFFSET ASCIIZ    ;asciiz specification
        INT     21H                 ;call dos
        MOV     HANDLE,AX           ;store file handle
;Read file's data into buffer
        MOV     AH,3FH              ;read function
        MOV     BX,HANDLE           ;file handle
        MOV     CX,10               ;No. bytes to read
        MOV     DX,OFFSET BUFFER    ;offset of buffer
        INT     21H                 ;call dos
;XOR the entire buffer
        MOV     CX,10               ;buffer count
        MOV     BX,OFFSET BUFFER    ;point to first byte
NEXT:   MOV     AL,[BX]             ;store byte in AL
        XOR     AL,CODE             ;get xor code
        MOV     [BX],AL             ;restore modified data
        INC     BX                  ;point to next byte
        LOOP    NEXT                ;process next byte
;Set file pointer to beginning of A:\TEST
        MOV     AH,42H              ;file pointer function
        MOV     AL,0                ;to beginning of file
        MOV     CX,0                ;       ''
        MOV     DX,0                ;       ''
        MOV     BX,HANDLE           ;file handle
        INT     21H                 ;call dos
;Write coded data to A:\TEST
        MOV     AH,40H              ;write function
        MOV     BX,HANDLE           ;file handle
        MOV     CX,10               ;No. bytes to write
        MOV     DX,OFFSET BUFFER    ;offset of buffer
        INT     21H                 ;call dos
;Close A:\TEST
        MOV     AH,3EH              ;close function
        MOV     BX,HANDLE           ;file handle
        INT     21H                 ;call dos
;Display data to screen
        MOV     AH,9
        MOV     DX,OFFSET PRINT
        INT     21H
        INT     20H                 ;return
CSEG    ENDS
        END     START
```

Figure 3. Assembler code for CODE.COM.

```
-W
Writing 000A bytes
-Q
A>TYPE TEST
AAAAAAAAAA
```

The utility to scramble A:TEST, CODE.COM, is listed in Figure 3. Here is how CODE.COM works: It opens the data file, saves the file handle, reads data into a buffer area, and XORs the data with a hexadecimal code (01-FFH) selected by the user. The file pointer is reset to the beginning of the file, the encoded data is written to A:TEST, the file is closed to finalize changes, and the modified data displayed on the screen. Before running, make sure TEST is stored in drive A and the caps-lock key is set.

```
A>CODE
SELECT XOR CODE (01-FF):   03
BBBBBBBBBB
```

To restore A:TEST to its original form, CODE.COM is executed a second time with the same XOR hex code.

```
A>CODE
SELECT XOR CODE (01-FF):   03
AAAAAAAAAA
```

CODE.COM is quite limited since it is only capable of scrambling one file (A:TEST) and anyone can access it. If the incorrect hex code is specified during the deciphering procedure, the data is scrambled a second time and made useless. To improve the worth of any data encryption utility, it must be able to scramble all data files, as well as approve authorized file access. This takes us to the next topic, the application of passwords.

Passwords

A competent file encryption utility needs the ability to recognize authorized users. Passwords are probably the most common means of identification. The computer requests an identifying password from the operator and compares his or her response to the correct password. If the keyed-in password and the one in memory match, access is allowed.

Password schemes are easy to implement; all they require is an input statement and a string comparison. A simple program that

demonstrates verification of a password is listed in Figure 4. The string "DECIPHER" was included in the data segment of PASSWORD.COM. If the user enters this password, the message "ALLOW ACCESS TO FILE" is displayed on the screen. Any other password displays "DENY ACCESS TO FILE." Convert PASSWORD.ASM to PASSWORD.COM and enter "DECIPHER" in caps.

```
A>PASSWORD
ENTER PASSWORD:   DECIPHER
ALLOW ACCESS TO FILE
A>PASSWORD
ENTER PASSWORD:   DECIPHE
DENY ACCESS TO FILE
```

For this example, the containment of the password was made easy because it was assembled with the rest of the program. But where in a file encryption scheme is the entered password contained? Let's consider a few options.

The most effective commercial security systems add extra ROM memory to your system so that a password is required for access even before the booting begins. But making a security ROM is difficult, requires special equipment, and is beyond the scope of most users.

Special sectors on the floppy or hard disk may be designated to store the password(s). This scheme could be developed to require either a single password for all files, or individual passwords. Of course, the sectors would have to be declared immediately after the disk format operation.

Another technique creates a special independent file, similar to the subdirectory file, for the purpose of containing file names and their corresponding passwords. This special file is permitted to grow to accommodate a maximum number of entries. Whenever a file is deciphered, the password information is removed from the special file.

```
;PASSWORD.ASM converts to PASSWORD.COM
;Demonstrates test of password
;Must enter password (DECIPHER) in CAPS
CSEG      SEGMENT
          ASSUME  CS:CSEG,DS:CSEG,ES:CSEG
          ORG     100H
START:    JMP     SKIP              ;skip data
MESS      DB      0AH,0DH,'ENTER PASSWORD:   $'
PASSWORD  DB      'DECIPHER'        ;stored password
ID        DB      9,10 DUP(0)       ;input password
ALLOW     DB      0AH,0DH,'ALLOW ACCESS TO FILE$'
DENY      DB      0AH,0DH,'DENY ACCESS TO FILE$'
;Display message to enter password
SKIP:     MOV     AH,9
          MOV     DX,OFFSET MESS
          INT     21H
;Input password
          MOV     AH,10
          MOV     DX,OFFSET ID
          INT     21H
;Compare ID to password
          MOV     SI,OFFSET PASSWORD  ;point to password
          MOV     DI,OFFSET ID+2      ;point to ID
          CLD                         ;clear directional flag
          MOV     CX,8                ;compare 8 bytes
          REPE    CMPSB               ;repeat if equal
          JZ      MATCH               ;jump, if match found
          MOV     DX,OFFSET DENY      ;''DENY ACCESS'' message
          JMP     PRINT
MATCH:    MOV     DX,OFFSET ALLOW     ;''ALLOW ACCESS'' message
PRINT:    MOV     AH,9                ;print message
          INT     21H                 ;call dos
          INT     20H                 ;return
CSEG      ENDS
          END     START
```

Figure 4. Assembler code for PASSWORD.COM.

Passwords may also be stored at the end of the encoded file. A few bytes are added when the data is scrambled, and then removed during the deciphering operation. I chose this technique for ENCRYPT.EXE, my customized encryption utility. But prior to the introduction of ENCRYPT.EXE, I will describe this password strategy.

The scheme begins with the loading of the entire data file into a 64K buffer within the utility program. The user inputs the password, up to 8 characters, and selects the option to scramble <S> or unscramble <U>. When a file is scrambled, the entered password is XORed and appended to the end of the file. A special 8-byte code immediately follows the encoded password. The special code is used to recognize files that were previously scrambled. The encrypted data, password, and special 8-byte code are written from the buffer to the data file. The scrambled file is now 17 bytes larger than the original data file.

When the data is deciphered, the password is entered and compared to the original password contained at the end of the file. If they agree, the entire file is read into the buffer area and decoded. The data file is truncated to zero length and the buffer area, minus the 17-byte appendix, is written to the file. This procedure permits data files to be scrambled/unscrambled any number of times without growing in size.

Introducing ENCRYPT

Many of the techniques described in this article have been assimilated into the file encryption utility offered in Figure 5. ENCRYPT.EXE, effective for files less than 64K in size, requires the user to select a password up to eight characters and an XOR hex code (01-FFH). ENCRYPT.ASM must be assembled into ENCRYPT.EXE at some time prior to execution.

To demonstrate ENCRYPT.EXE, A:TEST is scrambled once again. Check the contents of A:TEST for its original data:

```
A>TYPE TEST
AAAAAAAAAA
```

Load ENCRYPT and enter the path, file name, password, option, and XOR hex code as shown:

```
A>ENCRYPT

ENTER  DRIVE, PATH, FILE NAME:
(EXAMPLE:  A:\PATH\FILENAME.EXT)
A: EST

ENTER PASSWORD  (UP TO 8 CHARACTERS):   PASSWORD

PRESS:  <S> SCRAMBLE    <U>UNSCRAMBLE    <Q>QUIT
S
SELECT XOR CODE (01-FFH):   29

FILE SCRAMBLED
A>TYPE TEST
hhhhhhhhhhyhzz~f{m$AWBXCYDZ
```

After entering the file name (A\TEST), password (PASSWORD), option (S), and XOR hex code (29), the file is opened. The file size is determined (DOS Function 42H) and only files less than 64K are processed further. To resolve whether the data is already scrambled, the last 17 bytes of the file are read into a buffer area. If the special 8-byte code (AWBXCYDZ) is located, the file is closed and the program terminates. Whenever ENCRYPT.EXE terminates, one of these six messages indicates the status of the operation:

Figure 5. Assembler code for ENCRYPT.EXE.

```
;ENCRYPT.ASM converts to ENCRYPT.EXE
;Select XOR hex code (01-FF).
;Stores password at end of encrypted file.
;Input option and hex code in CAPS.
;*************************************************************
SSEG    SEGMENT  STACK
        DB       20 DUP  ('STACK   ')
SSEG    ENDS
;*************************************************************
DSEG    SEGMENT
FILE    DB   64,65 DUP (0)    ;asciiz string
PASSW   DB   9,10 DUP (0)     ;password
        DB   ''AWBXCYDZ''      ;special 8-byte EOF code
HANDLE  DW   ?                ;file handle
SIZEH   DW   0                ;file size high
SIZEL   DW   0                ; ''    ''   low
PCINTH  DW   0                ;file pointer high
POINTL  DW   0                ;  ''     ''    low
OPTION  DB   0                ;scramble,unscramble, or quit
BUFPC   DB   17 DUP(0)        ;password/code buffer
CODE    DB   0                ;XOR hex code
MESS1   DB   0AH,0DH,'ENTER DRIVE,PATH,FILE NAME:',0AH,0DH
        DB   '(EXAMPLE:  A:\PATH\FILENAME.EXT)',0AH,0DH,24H
MESS2   DB   0AH,0AH,0DH,'FILE SCRAMBLED',0AH,0AH,0DH,24H
MESS3   DB   0DH,0AH,0AH,'FILE NOT LOCATED',0AH,0AH,0DH,24H
MESS4   DB   0AH,0AH,0DH,'PRESS: <S>SCRAMBLE  <U>UNSCRAMBLE'
        DB   '  <Q>QUIT$'
MESS5   DB   0AH,0AH,0DH,'INCORRECT PASSWORD'
        DB   'OR FILE NOT SCRAMBLED',0AH,0AH,0DH,24H
MESS6   DB   0AH,0AH,0DH,'FILE ALREADY SCRAMBLED',0AH,0AH,24H
MESS7   DB   0AH,0AH,0DH,'FILE UNSCRAMBLED',0AH,0AH,0DH,24H
MESSH   DB   0DH,0AH,0AH,'SELECT XOR CODE (01-FF):   $'
MESSP   DB   0DH,0AH,0AH,'ENTER PASSWORD'
        DB   '  (UP TO 8 CHARACTERS):   $'
MESSB   DB   0DH,0AH,0AH,'FILE TOO BIG TO SCRAMBLE'
        DB   0AH,0AH,24H
DSEG    ENDS
;*************************************************************
ESEG    SEGMENT
CODE2   DB   0                ;XOR hex code
BUFFER  DB   0FFFEH DUP(''B'')   ;64K read/write buffer
ESEG    ENDS
;*************************************************************
CSEG    SEGMENT
MAIN    PROC   FAR
        ASSUME CS:CSEG,DS:DSEG,ES:ESEG,SS:SSEG

;Set return and DS/ES registers
START:  PUSH DS
        SUB  AX,AX
        PUSH AX
        MOV  AX,DSEG
        MOV  DS,AX
        MOV  AX,ESEG
        MOV  ES,AX

        CALL INPUT       ;input file/password/option
        MOV  AL,OPTION   ;check option for termination
        CMP  AL,''Q''
        JZ   QUIT        ;if Q, quit program
        CALL HEX         ;get hex code for xor
        CALL CRYPT       ;scramble/unscramble file
QUIT:   RET
MAIN    ENDP
;-------------------------------------------------------------
;Input file,path,password,and option to scramble/unscramble.
INPUT   PROC NEAR
        MOV  AH,9               ;display message
        MOV  DX,OFFSET MESS1    ;to enter file path
        INT  21H
        MOV  AH,0AH             ;input file path
        MOV  DX,OFFSET FILE
        INT  21H
        MOV  BX,OFFSET FILE+1   ;get string size and
        MOV  AL,[BX]            ;store in AL
        MOV  AH,0
        ADD  BX,AX              ;point to last byte
        INC  BX                 ;point to CR
        MOV  AL,0
        MOV  [BX],AL            ;store null character
        MOV  AH,9               ;display message to
        MOV  DX,OFFSET MESSP    ;enter password
        INT  21H
        MOV  AH,0AH             ;input password
        MOV  DX,OFFSET PASSW
        INT  21H
;Select option to scramble/unscramble/quit
REPEAT: MOV  AH,9               ;display option message
        MOV  DX,OFFSET MESS4
        INT  21H
        MOV  AH,8               ;input option
        INT  21H
        CMP  AL,''Q''             ;repeat if key other
        JZ   OK                 ;than Q, S, or U pressed
        CMP  AL,''S''
        JZ   OK
        CMP  AL,''U''
        JZ   OK
        JMP  REPEAT             ;repeat option message
OK:     MOV  OPTION,AL          ;save option
        RET
INPUT   ENDP
;-------------------------------------------------------------
;Input hex code for xor operation
HEX     PROC   NEAR
AGAIN:  MOV   DX,OFFSET MESSH     ;display code message
        MOV   AH,9
        INT   21H
        MOV   AH,1                ;input high digit
        INT   21H
        CMP   AL,30H              ;check for hex
        JB    AGAIN               ;if wrong, try again
        MOV   DL,AL               ;store high in DL
        SUB   DL,30H              ;change ascii to hex
        CMP   DL,9                ;compare with 9
        JBE   DIGIT               ;if less/equal, skip
        SUB   DL,7                ;subtract 7
        CMP   DL,9                ;check for hex
        JBE   AGAIN               ;if wrong, try again
        CMP   DL,16               ;check for hex
        JAE   AGAIN               ;if wrong try again
DIGIT:  MOV   CL,4
        SHL   DL,CL               ;shift DL 4 bits to left
        MOV   AH,1                ;input low digit
        INT   21H
        CMP   AL,30H              ;check for hex
        JB    AGAIN               ;if wrong, try again
        SUB   AL,30H              ;change ascii to hex
        CMP   AL,9                ;compare with 9
        JBE   DIGIT2              ;if less/equal, skip
        SUB   AL,7                ;subtract 7
        CMP   AL,9                ;check for hex
        JBE   AGAIN               ;if wrong, try again
        CMP   AL,16               ;check for hex
        JAE   AGAIN               ;if wrong, try again
DIGIT2: ADD   AL,DL               ;add high/low
        MOV   CODE,AL             ;store code in DS
        PUSH  DS
        MOV   DX,ES
        MOV   DS,DX
        MOV   CODE2,AL             ;store code in ES
        POP   DS
        RET
HEX     ENDP
;-------------------------------------------------------------
;Open the data file, check password/code, and
;perform xor operation on entire file.
CRYPT   PROC  NEAR
;Open the data file
```

(Figure 5 continued)

```
        MOV   AH,3DH      ;open file
        MOV   AL,2        ;read/write access
        MOV   DX,OFFSET FILE+2  ;point to asciiz
        INT   21H
        JNC   OPENED      ;jump if file opened
        MOV   AH,9        ;file not located
        MOV   DX,OFFSET MESS3
        INT   21H
        RET               ;quit program
OPENED: MOV   HANDLE,AX   ;save file handle
;Get file size
        MOV   AH,42H      ;move file pointer
        MOV   AL,2        ;to end of file
        MOV   CX,0
        MOV   DX,0
        MOV   BX,HANDLE   ;get handle
        INT   21H
        MOV   SIZEH,DX    ;save file size high
        MOV   SIZEL,AX    ; ''    ''    ''   low
        CMP   DX,0        ;check file size
        JE    CONTINUE    ;quit if file > 64K
        CMP   AX,65500    ;check file size
        JL    CONTINUE    ;quit if file > 64K
        MOV   AH,9              ;display message
        MOV   DX,OFFSET MESSB   ;''FILE TOO BIG''
        INT   21H
        JMP   CFILE       ;quit program
;Read  password/code
CONTINUE:
        MOV   AH,42H      ;reset file pointer
        MOV   AL,0        ;from start of file
        MOV   CX,SIZEH    ;file size high
        MOV   DX,SIZEL    ;file size low
        SUB   DX,17       ;point to password
        MOV   BX,HANDLE   ;get handle
        INT   21H
        MOV   AH,3FH      ;read file
        MOV   BX,HANDLE   ;get handle
        MOV   CX,17       ;No. of bytes to read
        MOV   DX,OFFSET BUFPC   ;point to pass/code buffer
        INT   21H
;Set ES same as DS
        PUSH  ES
        PUSH  DS
        POP   ES
;Check option for scramble
        MOV   AL,OPTION
        CMP   AL,''S''            ;for ''S'' option,
        JZ    SCRAM               ;jump to SCRAM
;Unscramble the file
        MOV   CX,9
        MOV   BX,OFFSET BUFPC   ;point to password
DECODE: MOV   AL,[BX]           ;decode the password
        XOR   AL,CODE           ;stored in scrambled file
        MOV   [BX],AL
        INC   BX
        LOOP  DECODE
;Verify password
        MOV   SI,OFFSET PASSW+2 ;point to input password
        MOV   DI,OFFSET BUFPC   ;point to original password
        MOV   CX,9
        CLD
        REPE  CMPSB             ;compare
        JZ    MATCH             ;if match,unscramble file
        MOV   AH,9              ;display
        MOV   DX,OFFSET MESS5   ;''INCORRECT PASSWORD''
        INT   21H
        POP   ES                ;restore ES
        JMP   CFILE             ;quit program
;Check special EOF code for previous scramble
SCRAM:  MOV   SI,OFFSET PASSW+11   ;point to EOF code
        MOV   DI,OFFSET BUFPC+9    ;point to end of file
        MOV   CX,8
        CLD
        REPE  CMPSB
        JNZ   MATCH             ;scramble if no match found
        MOV   AH,9              ;display message
        MOV   DX,OFFSET MESS6   ;''ALREADY SCRAMBLED''
        INT   21H
        POP   ES                ;restore ES
CFILE:                          ;close file
        MOV   AH,3EH            ;close function
        MOV   BX,HANDLE         ;file handle
        INT   21H
        RET                     ;quit file
;Read file into buffer
MATCH:  POP   ES                ;restore ES
        CALL  READ              ;read file into buffer
        JC    FAIL              ;quit if read failed
;XOR the entire buffer
        MOV   CX,SIZEL          ;get file size
        PUSH  DS
        MOV   DX,ES             ;set DS = ES
        MOV   DS,DX
        MOV   BX,OFFSET BUFFER  ;point to data
CRYPT1: MOV   AL,[BX]           ;byte into AL
        XOR   AL,CODE2          ;XOR byte
        MOV   [BX],AL           ;restore coded data
        INC   BX                ;point to next byte
        LOOP  CRYPT1            ;process next byte
        POP   DS
;Close file
        MOV   AH,3EH            ;close function
        MOV   BX,HANDLE         ;file handle
        INT   21H
;Truncate file to zero
        MOV   AH,3CH            ;truncate function
        MOV   CX,0              ;file attribute
        MOV   DX,OFFSET FILE+2  ;point to asciiz
        INT   21H
        MOV   HANDLE,AX         ;save new handle
        CALL  WRITE             ;write to file
        JC    FAIL              ;quit if operation failed
        CMP   OPTION,''U''        ;check option and
        JZ    UNMESS            ;point to unscramble message
        MOV   DX,OFFSET MESS2   ;''FILE SCRAMBLED''
        JMP   SMESS
UNMESS: MOV   DX,OFFSET MESS7   ;''FILE UNSCRAMBLED''
SMESS:  MOV   AH,9              ;display appropriate message
        INT   21H
        JMP   CLOSE
FAIL:   MOV   AH,9              ;display message
        MOV   DX,OFFSET MESS3   ;''FILE NOT LOCATED''
        INT   21H
        JMP   DONE
CLOSE:  CMP   OPTION,''U''        ;check option
        JZ    SKIP
        CALL  PASSWORD          ;if scramble, add password
SKIP:   MOV   AH,3EH            ;close file
        MOV   BX,HANDLE         ;get handle
        INT   21H
DONE:   RET
CRYPT   ENDP
;---------------------------------------------------------
;Reads data file into buffer
READ    PROC  NEAR
        MOV   AH,42H            ;reset file pointer
        MOV   AL,0              ;from start of file
        MOV   CX,0
        MOV   DX,0
        MOV   BX,HANDLE         ;get handle
        INT   21H
        MOV   AH,3FH            ;read file
        MOV   BX,HANDLE         ;get handle
        MOV   CX,SIZEL          ;No. of bytes to read
        PUSH  DS
        MOV   DX,ES             ;set DS=ES
        MOV   DS,DX
        MOV   DX,OFFSET BUFFER
        INT   21H
```

```
        POP   DS
        RET
READ    ENDP
;-------------------------------------------------------
;Writes buffer to data file
WRITE   PROC  NEAR
        MOV   AH,42H            ;reset file pointer
        MOV   AL,0              ;from start of file
        MOV   CX,0
        MOV   DX,0
        MOV   BX,HANDLE         ;get handle
        INT   21H
        MOV   AH,40H            ;write to file
        MOV   BX,HANDLE         ;get handle
        MOV   CX,SIZEL          ;No. of bytes
        CMP   OPTION,''S''        ;check option
        JZ    SKIP2             ;for unscramble, subtract
        SUB   CX,17
SKIP2:  PUSH  DS
        MOV   DX,ES
        MOV   DS,DX
        MOV   DX,OFFSET BUFFER
        INT   21H
        POP   DS
        RET
WRITE   ENDP
;-------------------------------------------------------
;Write password/code to end of data file
PASSWORD  PROC  NEAR
        MOV   CX,9
        MOV   BX,OFFSET PASSW+2
CRYPT2: MOV   AL,[BX]           ;encrypt password
        XOR   AL,CODE           ;with the same hex
        MOV   [BX],AL           ;code used to scramble
        INC   BX                ;data file
        LOOP  CRYPT2
        MOV   AH,40H            ;write to file
        MOV   CX,17             ;17 bytes (password/code)
        MOV   BX,HANDLE         ;get handle
        MOV   DX,OFFSET PASSW+2 ;point to password
        INT   21H
        RET
PASSWORD  ENDP
;-------------------------------------------------------
CSEG    ENDS
;*******************************************************
        END   START
```

```
    FILE SCRAMBLED
    FILE UNSCRAMBLED
    FILE NOT LOCATED
    FILE ALREADY SCRAMBLED
    FILE TOO BIG TO SCRAMBLE
    INCORRECT PASSWORD OR FILE NOT SCRAMBLED
```

If A:TEST was not previously scrambled, it is read into a buffer area and XORed with the selected value, in this case 29H. Next, A:TEST is truncated to zero (DOS Function 3CH) and the XORed data written to its file. Before closing A:TEST, the 17-byte appendix is written to the end of the file. The DOS TYPE command displays the contents of the scrambled file. The first 10 characters (hhhhhhhhhh) represent the encrypted data; the next 9 bytes (yhzz~f{m$) the XORed password and carriage return; and the last 8 bytes (AWBXCYDZ) the special code to identify a previously scrambled file.

To restore A:TEST, execute ENCRYPT.EXE again, but this time choose the <U> option to unscramble the file. If you enter the same password (PASSWORD) and XOR hex code (29), A:TEST is reinstated to its original data and size.

Summary

Now that you are familiar with file encryption techniques, you may wish to design your own system. Although, in theory, no encoding system is absolutely secure, you can make it extremely difficult to decipher files. For example, the algorithm to generate the data-scrambling pattern may be based upon the password. Thus, the user's password also serves as the unique key required to decipher the coded information. If "CRYPT" is chosen as the password, the file's data ("COMPUTER FILES ARE SECURE") is XORed in sequences of five as shown:

```
data:       COMPUTER FILES ARE SECURE
XOR with:   CRYPTCRYPTCRYPTCRYPTCRYPT
```

Bytes 1,6,11,16,21 are XORed with 43H (character "C"); bytes 2,7,12,17,22 with 52H (character "R"); and so on. You could make the password as large or small as you wish, but the only means to decipher the data is to know the unique algorithm.

Sometimes, a simple customized encryption system with no documentation may prove more effective than a highly sophisticated, well-documented utility. Assigning a deceptive file name and disguising what the program attempts to do may prove advantageous. For instance, refer to ENCRYPT as FLOWERS and have it display misleading messages during the execution:

```
A>FLOWERS

ENTER FLOWER

ENTER KEY

PRESS:  <C>CROSS  <U>UNCROSS  <Q>QUIT

CLASSIFICATION NUMBER:

''FLOWER CROSSED''
```

ENCRYPT is easy to use, does not modify the operating system, and requires no special installation. Also, the design of the password and XOR hex code is not difficult to remember. While ENCRYPT keeps out the casual snoop, it cannot deter sabotage in the form of unauthorized duplication or erasure of files. The complexity of your system should depend on the secrets you are protecting and from whom. ■

Data Base
Choosing Your Tools
by Art Carlson

The discussion of tools is a very emotional subject and it generates a lot of heated arguments. If you are ever with a group of programmers and the conversation drags, just start talking about your favorite word processor! Even the meek, quiet introverts will get loud and boisterous.

We all tend to settle on a favorite tool, and then use it for every conceivable application—whether or not it is the best tool for that situation. But, it is better to be familiar with several tools so that we can match them to the application. The choices are not simple, because they include many intangible factors, one of which is the programmer's temperment and working style.

There is no one DBMS which is best for all applications. Usually there is not even one DBMS which is absolutely the best choice for a given application. In the next few issues we'll look at some of the factors which should be considered when selecting the DBMS. Hopefully, readers will respond with their ideas and experiences.

Defining the Needs

I like to break the requirements into four areas: 1) The application, 2) The end user, 3) The developer, and 4) Anything else, which can include things such as time or money limitations. The requirements don't fit into these nice little boxes, because they overlap and are mutually dependent. I'll try to discuss each category separately, but at times we'll wander into other areas in order to consider how one decision will affect others.

Simple or Complex

A good place to start is to consider the size and complexity of the application. Is it simple or involved?

An example of a simple application could be a program which is used to print labels once a month for a small church mailing list.

An example of an involved application could be a point-of-sales (POS) package for a chain of retail stores.

The church program would contain a single data file without any references to any other data file, and would be set up as a single user system. This is called a *flat file*. This could be easily programmed using C, Pascal, or BASIC—you wouldn't have to use a formal DBMS, but there are easy to use flat file managers available. In this case the dominant factor will probably be minimum development time and expense. The speed at which it runs is not critical because it is only used once a month on a small list and the run time will be largely determined by the printer. A system crash or loss of the data files is an inconvenience, but not a disaster as long as there is a back up available.

The POS application can be very complex and demanding. Each store may have a number of registers with laser scanners, which requires multiuser capability. During peak periods the useage could reach 500 or more transactions per minute, and it will be used 12 or more hours per day. In this case the dominant factors are high speed performance, preserving the consistency of the database, and automatic recovery in the event of a crash. If the system goes down or can not keep up with the transactions on the Friday after Thanksgiving (frequently the busiest retail sales day of the year), it is a major disaster for the retailer. The loss of the transactions in progress in the event of a momentary power outage would merely mean that those items would have to be run back thru the scanner. However, the corruption of the data files would be extremely serious as it would shut the store down until the files could be restored.

In addition to tending the laser scanners and the registers, the POS system will have to keep track of the sales of each item to adjust the inventory, prepare daily sales summaries, record the efficiency of the register operators, balance the cash in the registers, record the sales activity during the day so that the proper number of register operators can be scheduled, etc. This requires multitasking (in addition to being multiuser for the registers), and may be too much for a single microcomputer to handle. There would probably be an embedded controller at each register, and I would probably go with several networked high-performance micros in the office to distribute the work load.

I realize that I am a rebel and that I refuse to do something just because everybody else is doing it, but I get very nervous about putting all the multiuser and multitasking functions on one CPU. I worked in product design and manufacturing, and I have seen what can happen to a production line when something goes down. A computer system for a production process or for POS just can not hang up or fail to keep up with the demand. The only acceptable down time is when there is a power outage, because all the other machinery and registers also stop—and then it had better come up running with no corruption of the files as soon as the power returns. These applications are much more demanding than the non-real-time office type systems, such as order processing, with which most programmers are familiar. If order processing goes down for an hour, you can alway work late or over the weekend to catch up. But, if a production line goes down you may have several hundred workers standing around drawing wages. If a POS system goes down, the sales are lost because the customers will leave and go somewhere else.

I do not feel that even a 25 MHz '386 system with the best operating system is good enough for some applications. I would rather install several networked systems and distribute the workload.

If there are six or eight stores in the chain, there will have to be provisions for reporting to the central office, and provisions for more extensive data analysis there.

Real Time or Batch

In the church application, they would probably pile up any changes or deletions and someone would update the file once a month. I prefer to keep the data entry person out of the main data files whenever possible in order to minimize the possibility of trashing the files—especially where they use volunteer help who only do the job occasionally. I also like to have transaction files which can be used to figure out what went wrong.

(Continued on page 25)

A Simple Multitasking Executive

Designing an Embedded Controller Multitasking System

by Richard Rodman

In embedded controller applications, the processor often must be shared among a number of dissimilar tasks (e.g. waiting for the user to press a key, sampling a number of remote sensor switches, waiting to send characters to a peripheral). There are several general approaches to the problem.

The first is to construct a large polling loop which performs all of these things in a kind of intermingled fashion. The drawback to this approach is that the logic is difficult to understand, and changes to one part of the system can cause seemingly unrelated parts of the system to stop working.

The second method, if all of the operations are more or less parallel and identical, is to construct a state machine. The drawback to this approach is that the task is broken into little pieces, much like sharing a novel among many people by tearing out all of the pages and passing them around, and the complexity can be difficult to comprehend.

The most elegant method is to write and debug each task without knowledge of the others, as a separate program, and then link them together with a multitasking executive. There are two basic "types" of multitasking. The first is called "preemptive" multitasking. Usually, a clock tick interrupts each task and moves execution to the next task. These interrupts are not synchronized with the task's execution. The second is called "non-preemptive" multitasking, where each task voluntarily surrenders the CPU at times when it is convenient. In most commercial operating systems, of course, both techniques are used at the same time.

When a task gives up the CPU (dispatching), the executive can decide the next task to run in a number of ways. The simplest way is to just run the tasks one after each other, "round-robin". Other methods involve multiple priority queues and other more complicated schemes, the end result of which is to tie up more and more of the CPU's time in the scheduler itself. Of course, in some real-time applications, you need to have certain tasks run as soon as some events occur.

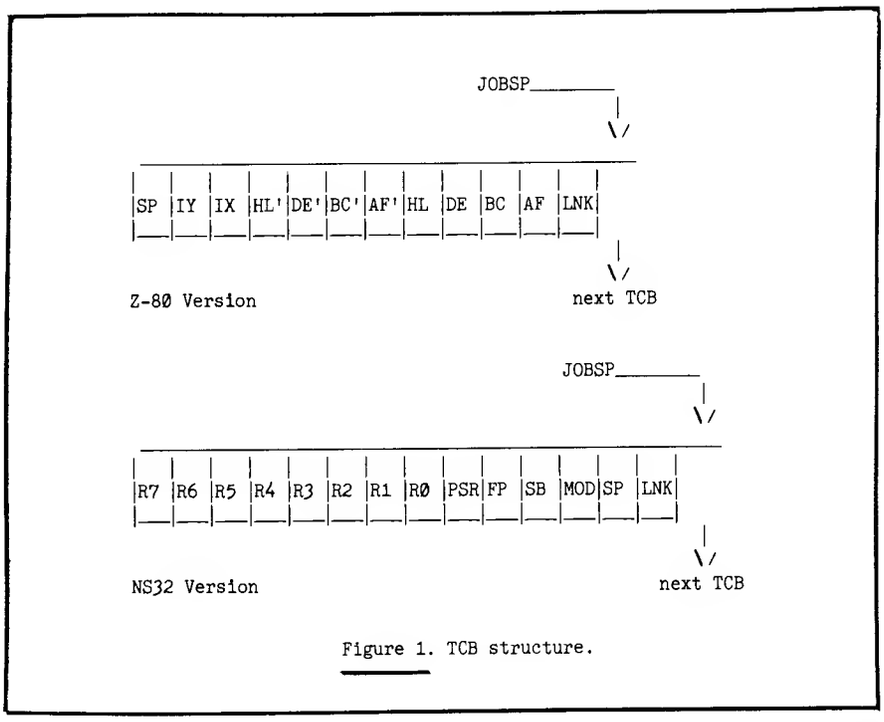


Figure 1. TCB structure.

Some executives described in the literature are completely equipped with routines for adding and deleting tasks, and are thus extensible to operating systems. Others switch tasks inflexibly and are more suited for small control applications with a known, small number of tasks. The executive presented here is one of the latter. Although the routines are not included, adding and deleting tasks can be accomplished by modifying the executive's task list. The executive has been optimized for task-switch time, which is as it should be—that is its main job. This requires the Task Control Block (TCB) to be designed somewhat around the CPU architecture, which could make adding and deleting tasks a little more difficult. Both Z-80 and NS32 (National Semiconductor 32016, 32032) versions are presented. The Z-80 version of the executive switches tasks in a consistent 446 T-states (112 microseconds at 4MHz), saving and restoring all registers.

The routine switches tasks by use of task control blocks which each point to the next task control block, and contain the register values (including stack pointer) for each task. The last task control block points back to the first one. In this way, to switch tasks, the executive has only to save all registers, pick up the next task control block, then restore all registers and return. The current task control block is pointed to by the memory location JOBSP. Figure 1 is a diagram of the task control block structure.

In some Z-80 applications it may not be necessary to save the alternate registers (perhaps they are used only within interrupt handlers), or it may not be necessary to save any registers at all. In such cases, great improvements can be made in switching speed.

In general, the programmer should identify "slack times" within each task wherein the task can relinquish the processor, and, at those points, call the entry point, call the location DISPAT. Care should be taken not to call DISPAT too often, or to consume too much CPU time in a task without calling DISPAT, as

this will prevent an even flow of control between the various tasks.

Alternatively, a periodic "tick" interrupt can be used to cause task switches. In such a case, the RET at the end of the executive must be replaced by a RETI instruction if Z80 peripherals are used (where the comment reads "all done"). If both voluntary and "tick"-initiated task switching is desired, the best thing to do is use a second copy of the executive with a RETI on the end for the "tick" interrupt. They can both share JOBSP and SAVSP.

The Z-80 executive itself is presented in Listing 1. Listings 2 and 3 present a three-task example which prints characters to the console of a CP/M system. As an exercise, try to figure out how many 1's and 2's will be printed. Later, try moving the calls to DISPAT around and seeing the effect on performance.

Listing 4 presents the equivalent multitasking executive and example tasks in NS32 code. This version uses 52-byte TCB's (although actually not all 52 bytes are needed). It does not save the floating-point registers.

The Z-80 is today probably the most commonly-used processor for embedded applications. The software tools are quite sophisticated, the CPU has sufficient computational power, and the cost is low. In some applications, though, the Z-80 runs out of steam. While some have considered the 80186, it is not significantly more powerful than the Z-80. Larger embedded systems commonly employ 68000 or NS32 (National Series 32000) family processors. However, it is difficult to construct a low-cost 68000-family system, even with the 68008. The NS32 family has full object code compatibility across all CPU's, meaning that no software changes are necessary if it is necessary to move to a more powerful CPU. Besides, the code is easier to write in the first place. After studying the three, I decided I liked programming in NS32 assembler the best.

When reading the listings, remember that while the Z-80 moves from right to left (Intel style), the NS32 moves from left to right (National and Motorola style).

REFERENCES

1. *Multitasking Scheduler works without DOS*, David M. Howard, *EDN*, September 15, 1982, page 194.

2. *Add Multiple Tasks to your Communication and Control Program*, Jerry Holter, *BYTE*, September 1983, page 445.

3. *A Kernal (sic) for the MC68000*, Steve Passe, *Dr. Dobb's Journal*, November 1983, page 20.

4. *Non-Preemptive Multitasking*, Joe Bartel, *The Computer Journal*, issue #30 (1988), page 37.

---LISTING ONE---

```
;---Multitasking Executive---

; Each  job has a stack area of 22 bytes for storing  registers.
; Above this, and pointed to by JOBSP for each task, is a pointer
; to the next task in round-robin order.
; Call here to relinquish processor to next task

DISPAT:
     DI
     LD   (SAVSP),SP
     LD   SP,(JOBSP)
     PUSH AF
     PUSH BC
     PUSH DE
     PUSH HL          ;save primary registers
     EX   AF,AF'
     EXX
     PUSH AF          ;save alternate registers
     PUSH BC
     PUSH DE
     PUSH HL
     PUSH IX          ;save index registers
     PUSH IY
     LD   HL,(SAVSP)
     PUSH HL          ;save stack pointer

; Cold start entry point

DISPA0:
     LD   SP,(JOBSP)  ;pick up pointer to next job
     POP  HL          ;get address of next job
     LD   (JOBSP),HL  ;save in memory
     LD   DE,-22      ;offset to bottom of stack
     ADD  HL,DE
     LD   SP,HL       ;put into SP to pop registers
     POP  HL
     LD   (SAVSP),HL  ;get stack pointer first
     POP  IY
     POP  IX          ;restore index registers
     POP  HL
     POP  DE
     POP  BC
     POP  AF          ;restore alt registers
     EXX
     EX   AF,AF'
     POP  HL
     POP  DE
     POP  BC
     POP  AF          ;restore pri registers
     LD   SP,(SAVSP)
     EI
     RET              ;all done
SAVSP:
     DEFS 2           ;temp storage location
JOBSP:
     DEFS 2           ;pointer for this job
```

---LISTING TWO---

```
; Task control blocks.  Notice how each task block points to the
; next one, and the last one loops around to the first.

     DEFW TSK1SP      ;stack pointer
     REPT 10
     DEFW 0
     ENDM
TSK1:
     DEFW TSK2        ;link to next process

     DEFW TSK2SP      ;stack pointer
     REPT 10
     DEFW 0
     ENDM
TSK2:
     DEFW TSK3        ;link to next process

     DEFW TSK3SP      ;stack pointer
     REPT 10
     DEFW 0
     ENDM
TSK3:
     DEFW TSK1        ;link to next process

     DEFS 24
TSK1SP:
     DEFW TASK1

     DEFS 24
TSK2SP:
     DEFW TASK2

     DEFS 24
TSK3SP:
     DEFW TASK3
```

---LISTING THREE---

```
; Entry point for the three-task example.

     LD   HL,TSK1
     LD   (JOBSP),HL  ;set index to task one
     JP   DISPA0

; Actual code for the tasks.

TASK1:
     LD   E,'1'
     LD   C,2
     CALL 5           ;print a 1
     LD   E,32
TASK1A:
     LD   B,0
     DJNZ $
     CALL DISPAT
     DEC  E
     JP   NZ,TASK1A
     JP   TASK1

TASK2:
     LD   E,'2'
     LD   C,2
     CALL 5           ;print a 2
     LD   E,8
TASK2A:
     LD   B,0
     DJNZ $
     CALL DISPAT
     DEC  E
     JP   NZ,TASK2A
     JP   TASK2

TASK3:
     LD   E,128
TASK3A:
     LD   B,0
     DJNZ $
     CALL DISPAT
     DEC  E
     JP   NZ,TASK3A
     JP   0           ;stop the whole mess
```

--- LISTING FOUR ---

This NS32 version can be assembled with Neil Koozer's Z32 assembler and will run under the TDS or SRM monitors.

```
MODBASE

;The task control blocks are filled in at run-time in the NS32
;version.  This is because position-independent code is always
;used.

        DS      52
TSK1    DS      4       ;link to next process
        DS      52
TSK2    DS      4
        DS      52
TSK3    DS      4

        DS      32      ;stack areas
TSK1SP  DS      4
        DS      32
TSK2SP  DS      4
        DS      32
TSK3SP  DS      4

SAVSP   DS      4               ;temp storage location
JOBSP   DS      4               ;pointer for this job
ORIGSP  DS      4               ;original stack pointer

        DD      $-MODBASE       ;pseudo module table entry for SRM
        DD      LINKBEG-MODBASE
        DD      CODEBEG-MODBASE
        DD      CODEEND-MODBASE
LINKBEG
CODEBEG
        BR      START:W

; --- Multitasking Executive ---

;Each job has a stack area of 52 bytes for storing registers.
;Above this, and pointed to by JOBSP for each task, is a pointer
;to the next task in round-robin order.
;Call here to relinquish processor to next task

;R7 R6 R5 R4 R3 R2 R1 R0 PSR FP SB MOD SP NEXT
;52 48 44 40 36 32 28 24 20  16 12 8   4  0

DISPAT
        SPRD    SP,SAVSP        ;save current stack
        LPRD    SP,JOBSP        ;get job stack

        MOVD    SAVSP,TOS       ;save stack register
        SPRD    MOD,TOS         ;save mod register
        SPRD    SB,TOS          ;save static base
        SPRD    FP,TOS          ;save frame pointer
        SPRD    UPSR,TOS        ;save UPSR
        SAVE    [R0,R1,R2,R3,R4,R5,R6,R7]

; Cold start entry point

DISPA0
        MOVD    JOBSP,R0        ;pick up pointer to next job
        MOVD    0(R0),R0        ;get address of next job
        MOVD    R0,JOBSP        ;save in memory
        ADDR    -52(R0),R0      ;offset to bottom of stack
        LPRD    SP,R0           ;put into SP to pop registers
        RESTORE [R0,R1,R2,R3,R4,R5,R6,R7]
```

```
        LPRD    UPSR,TOS        ;restore psr
        LPRD    FP,TOS          ;restore frame pointer
        LPRD    SB,TOS          ;restore static base
        LPRD    MOD,TOS         ;restore mod register
        LPRD    SP,TOS          ;restore stack pointer
        RET     0               ;and return
.pa
;Entry point for the three-task example.
;Because the NS32 code is position-independent, the actual addresses have to
;be calculated at run-time.

TEST    DB      'TEST',13,10

START
        SPRD    SP,ORIGSP       ;save the original stack pointer

        ADDR    TEST,R1
        MOVQD   6,R2
        MOVQD   0,R3
        MOVQD   4,R0
        SVC

;We need to init the values of the special registers in the TCB's so that
;invalid stuff doesn't get loaded into the real CPU registers.

        ADDR    TSK1,R0         ;point to task descriptor for task 1
        ADDR    TSK1SP,-4(R0)   ;put addr of stack into TCB
        SPRD    MOD,-8(R0)      ;set mod reg
        SPRD    SB,-12(R0)      ;set sb reg
        SPRD    FP,-16(R0)      ;set fp
        SPRD    UPSR,-20(R0)    ;set upsr
        ADDR    TSK2,0(R0)      ;link it to task 2

        ADDR    TSK2,R0         ;point to task 2 TCB
        ADDR    TSK2SP,-4(R0)   ;set sp reg
        SPRD    MOD,-8(R0)      ;set mod reg
        SPRD    SB,-12(R0)      ;set sb reg
        SPRD    FP,-16(R0)      ;set fp
        SPRD    UPSR,-20(R0)    ;set upsr
        ADDR    TSK3,0(R0)      ;link it to task 3

        ADDR    TSK3,R0         ;point to task 3 TCB
        ADDR    TSK3SP,-4(R0)   ;set sp reg
        SPRD    MOD,-8(R0)      ;set mod reg
        SPRD    SB,-12(R0)      ;set sb reg
        SPRD    FP,-16(R0)      ;set fp
        SPRD    UPSR,-20(R0)    ;set upsr
        ADDR    TSK1,0(R0)      ;close the loop back to task 1

        ADDR    TASK1,TSK1SP    ;prepush start of task on stack
        ADDR    TASK2,TSK2SP
        ADDR    TASK3,TSK3SP

        ADDR    TSK1,JOBSP      ;set index to task one
        BR      DISPA0
```

```
; -- Task 1 --
;Task 1 prints a 1 every so often.

HI1     DB      '1'

TASK1
        ADDR    HI1,R1          ;pointer to string
        MOVQD   1,R2            ;length of string
        MOVQD   0,R3            ;channel number
        MOVQD   4,R0            ;service 4, write a string
        SVC

        MOVB    32,R6           ;32 dispatches
TASK1A
        MOVQB   0,R5
TASK1B
        ACBB    -1,R5,TASK1B    ;little delay
        BSR     DISPAT          ;dispatch
        ACBB    -1,R6,TASK1A    ;32 dispatch loop
        BR      TASK1           ;go back and print message

; -- Task 2 --
;Task 2 prints a number 2 four times as fast as task 1.

HI2     DB      '2'

TASK2
        ADDR    HI2,R1
        MOVQD   1,R2
        MOVQD   0,R3
        MOVQD   4,R0
        SVC
        MOVB    8,R6
TASK2A
        MOVQB   0,R5
TASK2B
        ACBB    -1,R5,TASK2B    ;little delay
        BSR     DISPAT          ;dispatch
        ACBB    -1,R6,TASK2A    ;loop 8 times
        BR      TASK2           ;go back and print message

; -- Task 3 --
;Task 3 waits a while (dispatching, of course), and then shuts down the
;whole thing.
TASK3
        MOVB    128,R6
TASK3A
        MOVQB   0,R5
TASK3B
        ACBB    -1,R5,TASK3B    ;little delay
        BSR     DISPAT          ;dispatch
        ACBB    -1,R6,TASK3A

        LPRD    SP,ORIGSP       ;restore orig sp
        RXP     0               ;return to monitor
CODEEND
        END     ■
```

CP/M's Magical IOBTYE

Using Tables for Space Efficient Implementation

by Donald C. Kirkpatrick

Don Kirkpatrick has been designing hardware and software since he graduated from college 19 years ago. He has a Ph.D. in electrical engineering and is a registered professional engineer. He works for Tektronix designing logic analyzers. He built his first CP/M system cira 1980 and has been improving it ever since. In his spare time, he teaches a graduate class in switching and automata for Oregon State University. He can be reached via electronic mail at the following net addresses:

donk@dadla.LA.TEK.COM

UUCP: {ihnp4 | decvax | ucbvax}!
tektronix!dadla!donk

ARPA: donk%dadla.LA.TEK.COM
@RELAY.CS.NET

I still remember my feeling of despair as the realization of what would be required for a full IOBYTE implementation slowly seeped into my mind. I was writing my first CP/M BIOS, with my Digital Research CP/M 2.2 Alteration Guide on my left, my Z80 Assembler Manual on my right, and my new terminal before me. I could see no way of implementing the functionality described in the Alteration Guide without utterly abandoning my BIOS space budget. Fortunately, first impressions are often deceiving. A full IOBYTE implementation need not be the multiheaded monster I first envisioned.

How can one byte take up so much space? It can't, but if not managed properly, the code required to make use of that byte can consume an inordinate amount of space. The major disadvantage to implementing the IOBYTE has always been that a complete implementation requires a very large fraction of the space available for the BIOS. A typical implementation of two terminals and two printers can consume about ⅓ of the entire space normally allocated to the BIOS, so its implementation has always been optional. Many systems just ignore the IOBYTE. Presented here is a remarkably space efficient method to completely implement the IOBYTE. This method also has the advantage of greatly simplifying the task of adding new device drivers. Each new physical device is added by changing one byte of code in a table and adding five to fifteen bytes of new code.

The Function of the IOBYTE

The purpose of the IOBYTE is to make it easy to redirect input or output between several devices. Suppose, for example, you have two printers. You could buy a switch box or unplug one and plug in the other every time you switch between them. With the IOBYTE implemented, you have a third option; the value of the IOBYTE determines which printer receives the output. Changing printers is as easy as typing a STAT command.

The CP/M has four logical input/output devices: CONSOLE, READER, PUNCH, and LIST. Also, there are 12 physical input/output devices: TTY:, CRT:, BAT:, UC1:, RDR:, UR1:, UR2:, PUN:, UP1:, UP2: LPT:, and UL1:. The sole function of the IOBYTE, which is found at address 0003H in memory, is to inform the BIOS how to map the four logical devices to the 12 physical devices. To make it easier to interpret the mapping algorithm, the IOBYTE may be thought of as four fields, one field for each logical device. The bit pattern in each field determines which physical device will be active when the logical device that owns that field is called upon for input or output. Since each field contains two bits, each logical device can be assigned four different ways. The fields and their values are as shown in Figure 1.

If the IOBYTE is implemented, whenever any logical device is addressed, the BIOS examines the appropriate field in the IOBYTE. Based on the bit pattern in that field, the request is directed to the proper physical device.

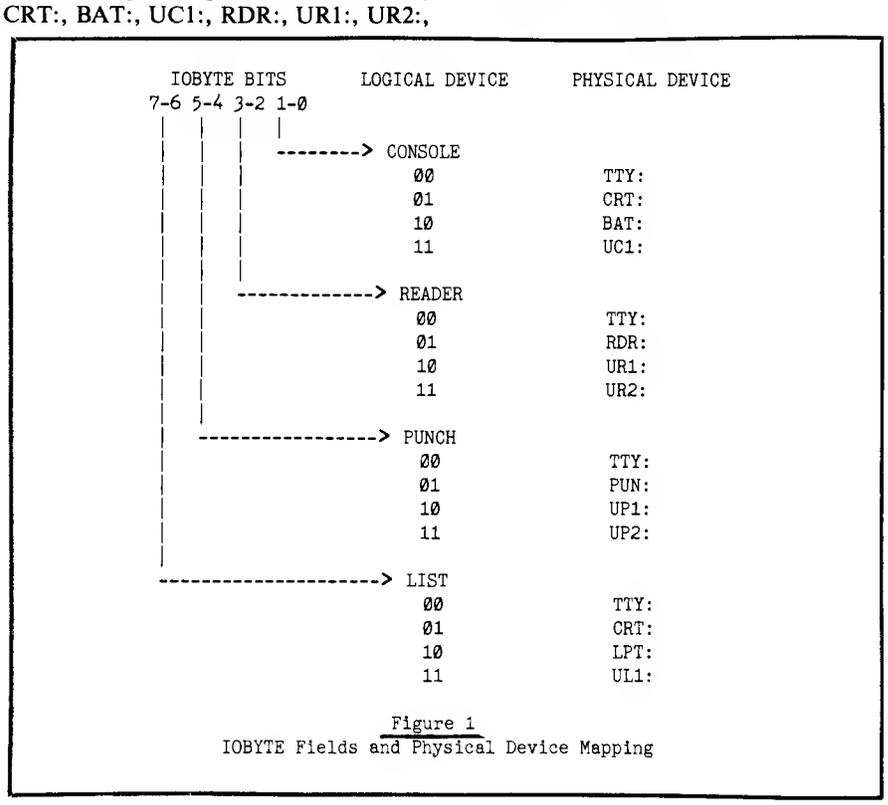


Figure 1
IOBYTE Fields and Physical Device Mapping

The device BAT: is not really a device at all. Rather, BAT is an abbreviation for batch. If the BAT: encoding appears in the CONSOLE field of the IOBYTE, then all CONSOLE reads and writes are redirected to the logical READER and LIST devices. When the READER or LIST receive the redirected console requests, they in turn look to the IOBYTE a second time to determine the actual physical driver to process the request.

The IOBYTE is initialized by the BIOS during the cold boot. Warm boots (^C) do not change the IOBYTE. Rather, device assignments are changed via the STAT utility.

Input/output redirection can be very handy. For example, a friend brought over her CP/M machine one day for a file transfer session. When it came time to connect the serial ports from the two machines, her system had only one cable with one kind of connector. No problem, we just plugged that connector into the serial port on my machine with the proper mating connector and redirected my console to that serial port. All of a sudden, her machine was the console for my machine.

Does Your System Already Implement The IOBYTE?

One reason for the success of CP/M is it can be used with many different hardware configurations. CP/M is organized so all hardware dependent code resides in the BIOS. It is the responsibility of the person writing the BIOS to implement the IOBYTE. You can determine if your system uses the IOBYTE by running a simple test. The value of IOBYTE can be displayed and changed by the standard CP/M transient command STAT. To display the current logical to physical mapping, type:

```
STAT DEV:
```

and you should see something like the following:

```
CON: IS TTY:
RDR: IS TTY:
PUN: IS TTY:
LST: IS TTY:
```

Your actual display depends upon what your BIOS IOBYTE default selections are, but many systems just set the IOBYTE to zero and then ignore it, so the above display may be quite typical. To see if you have the IOBYTE implemented, change the LST: device to the same device as the console (if it is not already):

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

and then type a control P. If the IOBYTE is implemented, then for each character typed, two will appear on your display. If the second character goes to your regular print device, the IOBYTE is not implemented.

What Makes the IOBYTE so difficult?

Most example implementations involve a great number of loads, mask operations, tests, and conditional jumps (see for example, *Inside CP/M* by David E Cortesi, pp. 258-260). Even a carefully written implementation involving only four physical devices can easily exceed 200 bytes. A programmer pressed for space is likely to write cryptic code. We all know such code is likely to contain errors and can be very difficult to debug. A better way is needed.

When an implementation makes use of regular structures such as tables, it is much easier to update and change as new features are added. My method of implementing the IOBYTE uses three regular structures: a standardized form for the five BIOS serial input/output calls, a table of addresses mapping a call to the proper input/output hardware driver, and a standardized form for the driver itself. Using common structures for these routines results in dramatic space savings over the random test and jump method normally given in IOBYTE implementation examples. Changes and additions are trivial using the structures presented here.

The code listed in the following figures is the actual code from my BIOS. I have split the listing into separate figures for clarity, but the code in each figure is separated from the code in the adjacent figure only by a PAGE pseudo-op.

Logical and Physical Device Drivers

All byte oriented input/output is performed via calls to five BIOS routines: CONIN, CONOUT, READER, PUNCH, and LIST. In addition, there are two routines to return the status of the console and list devices: CONIST and LISTST. These seven routines are mapped to fifteen physical devices, three console devices and four reader, punch, and list devices. There is also the special console BAT: option for a double layer of input/output redirection.

You probably do not have fifteen different physical input/output devices. On my system there are just three serial ports and one parallel port. These four physical devices are repeated in the mapping. For example, the console driver can be mapped to the three serial devices while the list driver can be mapped to all four physical devices. I have chosen names for my physical devices to make it easier for me to remember which is which. These names are: TTY, CRT, and DTE for the three serial devices and LST for the parallel port. It is easier to remember than the Digital Research chosen generic names such as UR1.

```
CONOUT: CALL    CONOST          ; console output
        JR      Z,CONOUT        ; output device not ready
        JR      OUTPUT          ; go output character

READER: CALL    READST          ; reader input
        JR      Z,READER        ; no input yet
        DEC     L               ; get port number
        LD      C,(HL)          ; into c register
        IN      A,(C)           ; get character
        RET                     ; reader returns all eight bits

PUNCH:  CALL    PUNST           ; punch output
        JR      Z,PUNCH
        JR      OUTPUT

LIST:   CALL    LISTST          ; list output
        JR      Z,LIST
OUTPUT: DEC     L               ; point HL at port number
        LD      A,C             ; save output character
        LD      C,(HL)          ; port number to c reg
        OUT     (C),A           ; and output
        RET

CONIN:  CALL    CONIST          ; console input
        JR      Z,CONIN
        DEC     L
        LD      C,(HL)
        IN      A,(C)
        AND     7FH             ; console returns only 7 bits
        RET
```

Figure 2
Logical Input/Output Driver Routines

All logical device routines have the same structure. First, the device calls a status routine to determine whether or not the physical device is able to perform the indicated operation. If the device is not ready, the driver repeatedly calls the status routine until the operation is possible. Next, the requested operation is performed. Lastly, the device returns to the caller.

One key to space efficiency is to note that once a physical device is ready, only a port number is required to complete the operation. Knowing which physical device is responding with "ready" is not information the logical driver needs to know. An important space saving fact emerges from this observation: the entire mapping task can be isolated to the physical status routines. We only need to make the status routines return the port number along with the wait/ready status.

Status Subroutines

Thus, we see the burden of determining the logical to physical mapping falls upon the status routines. This makes sense when you consider that two of the status routines (CONIST and LISTST) must themselves be callable from CP/M. The real breakthrough comes when the algorithm for performing the mapping is reduced to a small, simple table of twenty bytes. No more test and jump code! This table contains one entry per bit pattern for each of the five I/O status routines (CONIST, CONOST, and so on).

It is important to have an efficient method of finding an entry in the table. Every status routine calls the same table search subroutine. A status routine provides two calling parameters: the number of times to rotate the IOBYTE to bring the appropriate bit field to the two least significant bit positions and an offset into the table so the proper block of four status routine entries is addressed. The SEARCH subroutine finds the proper table entry and jumps to the hardware dependent ready routine. By forcing all of the IOBYTE routines to be in the same page (same high address byte) the table need only contain the lower byte of the ready routine address.

The easiest way to return the port address is to make use of the ready routine address left in the HL register when SEARCH jumps to the ready routine. Suppose the byte before the entry point to the ready routine is the port address for performing the indicated operation. If only status is needed, the HL register is discarded by the calling routine. But when input/output is required, the calling routine need only decrement HL and load the C register using the HL as a pointer. Armed with the port number in the C register, the transfer can be completed

```
TABLE:

; Console input mapping
        DEFB    LOW TTYST               ; conist tty
        DEFB    LOW CRTST               ; conist crt
        DEFB    LOW READST              ; conist bat
        DEFB    LOW DTEST               ; conist dte

; Console output mapping
        DEFB    LOW TTYOST              ; conost tty
        DEFB    LOW CRTOST              ; conost crt
        DEFB    LOW LISTST              ; conost bat
        DEFB    LOW DTEOST              ; conost dte

; Reader input mapping
        DEFB    LOW TTYST               ; readst tty
        DEFB    LOW CRTST               ; readst crt
        DEFB    LOW TTYST               ; readst tty
        DEFB    LOW DTEST               ; readst dte

; Punch output mapping
        DEFB    LOW TTYOST              ; punst tty
        DEFB    LOW CRTOST              ; punst crt
        DEFB    LOW LPTST               ; punst lpt
        DEFB    LOW DTEOST              ; punst dte

; List output mapping
        DEFB    LOW TTYOST              ; listst tty
        DEFB    LOW CRTOST              ; listst crt
        DEFB    LOW LPTST               ; listst lpt
        DEFB    LOW DTEOST              ; listst dte
```

Figure 3
IOBYTE Cross Reference Table

```
CONIST: LD      HL,800H + LOW TABLE     ; console input status
        JR      SEARCH

CONOST: LD      HL,804H + LOW TABLE     ; console output status
        JR      SEARCH

READST: LD      HL,208H + LOW TABLE     ; reader status
        JR      SEARCH

PUNST:  LD      HL,40CH + LOW TABLE     ; punch status
        JR      SEARCH

LISTST: LD      HL,610H + LOW TABLE     ; list status

; enter search with:
;       H --> amount to shift iobyte to move proper bits to 0-1
;       L --> amount to add to iobyte bits to index into adr table
; all iobyte code must be in the same page

SEARCH: LD      A,(IOBYTE)              ; get iobyte
SRCH1:  RRCA                            ; shift
        DEC     H                       ; done?
        JR      NZ,SRCH1                ; not yet
        AND     00000011B               ; mask off undesired bits
        ADD     A,L                     ; add table displacement index
        LD      L,A                     ; to form low table address byte
        LD      H,HIGH TABLE            ; load high address byte
        LD      L,(HL)                  ; extract address of routine
        JP      (HL)                    ; go to ready routine
```

Figure 4
Search Routine To Find Physical Device Ready Routine

```
;       Input ready routines
        DEFB    SIO+1                   ; tty input port
TTYST:  IN      A,(SIO+3)               ; get tty input status
        JR      SETØ

        DEFB    UARTD                   ; crt input port
CRTST:  IN      A,(UARTC)               ; get crt input status
        RLA
        RLA
        SBC     A,A
        RET

        DEFB    SIO+Ø                   ; dte input port
DTEST:  IN      A,(SIO+2)               ; get dte input status
        JR      SETØ

;       Output ready routines
        DEFB    SIO+Ø                   ; dte output port
DTEOST: IN      A,(SIO+2)               ; get dte output status
        JR      SET2

        DEFB    SIO+1                   ; tty output port
TTYOST: IN      A,(SIO+3)               ; get tty output status
        JR      SET2

        DEFB    PRDATA                  ; printer parallel port
LPTST:  IN      A,(PRSTAT)              ; get printer output status
        CPL
        JR      SETØ

        DEFB    UARTD                   ; crt output port
CRTOST: IN      A,(UARTC)               ; get crt output status
        RLA
        JR      NC,CRTST1
        CPL
SET2:   RRA
        RRA
SETØ:   RRA
CRTST1: SBC     A,A
        RET

        IF      HIGH $ - HIGH CONOUT    ; Force an assembly error
        *** I/O routines must be in one page ***
        ENDIF
```

Figure 5
My Hardware Dependent Ready Routines

with just an IN A,(C) or OUT (C),A instruction, completely independent of which physical device responded "ready".

Notice the only hardware specific code is the portion to determine the ready status. If all driver routines use the same ready convention as CP/M requires, then the status routines can be called by either a driver or CP/M, again doing double duty. The convention is: the accumulator contains zero if the device is unable to perform input/output and 0FFH if the device is ready. This permits the CONIN, READER, and LIST routines to use CONIST and LISTST exactly the same as any other CP/M routine.

My hardware dependent ready routines are shown in Figure 5. Clearly, you do not have the same port numbers or bit numbers for your UARTs as I. But just this little bit of code is all you have to rewrite to implement this IOBYTE scheme. Notice even here in the hardware ready routines, commonality is used to good advantage. By rotating the proper bit from each port into the carry, making the accumulator 0 or 0FFH is accomplished by one of two SBC instructions, no matter which physical device is checked.

Customizing STAT

Names should be easy to remember and meaningful. Terms like UP1 and UR2 can be cryptic. You may find it difficult to remember which device corresponds to which mnemonic. A much better way is to use an acronym meaningful to you. Now that you have a full IOBYTE implementation, why not customize STAT to reflect the meaningful names you have chosen for your physical devices? At the start of STAT is a table to control which acronym corresponds to each bit pattern. This table begins at address 0159H and consists of sixteen entries of four alphanumeric characters per entry. These entries can be easily changed using DDT, SID, or ZSID to whatever you choose. After you have completed your customization, you can check your work by typing:

```
STAT VAL:
```

and you will receive a complete display of your modified IOBYTE selections.

Try a full IOBYTE implementation. I'll bet you will find the space consumed a bargain when compared to the increased usefulness of your system. ■

The ZCPR3 Corner

by Jay Sage

For this issue I am going to live up to a long-standing tradition: once again I am not going to cover the material that I said I was. Last time I presented half the new material on ARUNZ and said I would cover the second half this time. Well, I am not going to. ARUNZ is now up to version 'N' (it was 'J' last time). Until it settles down a bit, it is probably futile to try to describe it. Besides that, I am getting a little bored with the subject (though obviously not with the program), and perhaps you are, too.

Though living up to tradition, I am going to reverse a trend. For some time now my columns have been getting longer and longer. This time I really am going to write a short one. Besides the fact that because of me Art Carlson is apparently running low on printer's ink, I am just about written out, having just completed the manuals for NZ-COM and Z3PLUS.

NZ-COM and Z3PLUS

Those manuals started out being simple affairs, but I just don't seem to be able to get my obsession with thoroughness and completeness under control, and they soon turned into full-fledged books about the respective product and Z-System in general. I've been burning the midnight (actually, 2 a.m.) oil for the past two or three months. Each manual now runs about 80 pages! No wonder I don't have many words left in my system at this point.

Though somewhat reluctant to indulge in self-praise, I have to say that the manuals are really quite good, and the products (NZ-COM and Z3PLUS) are absolutely fantastic. Joe Wright (NZ-COM), Bridger Mitchell (Z3PLUS), and I have had a very enjoyable and highly productive partnership in this effort. I sincerely urge you all to buy the automatic, universal, dynamic Z-System appropriate for your computer: NZ-COM for CP/M-2.2 computers and Z3PLUS for CP/M-Plus computers. Both are $69.95 from Sage Microsystems East, Plu*Perfect Systems, or Alpha Systems (see ads in TCJ).

After the experience bringing these products to market, I will no longer laugh so heartily when I hear stories about Borland or Lotus or Ashton-Tate not delivering their products on the promised dates. Hopefully you have lost your TCJ issue #32 and have forgotten that I wrote there, and I quote: "By the time you read this, they will definitely be available." In issue #33 I said, "the two new dynamic Z-Systems that will have been released by the time you read this." That almost made a double liar out of me. Luckily, issue #33 was delivered just enough behind schedule to let that statement squeak through—barely.

With respect to NZ-COM, Joe Wright and I have agreed to publicly blame each other for the delay. Actually, following common practice, we originally both agreed to blame Bridger Mitchell, though he, of course, had nothing to do with the NZ-COM delay (Z3PLUS is another story). However, now that Bridger has a TCJ column, too, we worried that such a slander might not go unanswered. Anyway, Joe can blame me for not getting the manual done on time, and I can blame Joe for not writing the code to conform to my description of it in the manual! You can readily see that no one should ever get involved in a product development all by himself. Always make sure there is someone else to blame.

The truth of the matter is that we really thought the coding was complete by early April and that a simple manual could be knocked off in a week (naive!!). In fact, as I alluded to above, the scope of the manual kept expanding. At the same time, as we attempted to describe the programs very precisely, we discovered a number of deficiencies in the code. Some coding limitations that we thought we would accept in the current version of NZ-COM and would upgrade later just didn't seem acceptable any more once we wrote them down on paper. As a result, we have really skipped version 1 of NZ-COM and have gone directly to version 2. There are quite a few exciting new features beyond what I described in issue #32; many will appeal especially to those with a penchant for the unconventional (but I won't say any more about them now).

PRL Files and Type-4 Programs

One of the new features introduced with ZCPR34 is the type-4 program. A number of questions have been appearing in messages on Z-Nodes, so I thought I would say a few words on this subject.

Just to refresh your memory, ordinary CP/M program files are loaded beginning at an address of 100H. This was also true of Z-System programs. They differ from standard CP/M programs in that the code starts with a special header. One item in the header is the text string 'Z3ENV' beginning with the fourth byte of the program. This string is used to identify the program as a Z-System program.

After the text string comes a number, now called the program type identifier. If the number is 2, then the so-called environment descriptor is included in the first page of the program file. These type-2 programs are rarely seen today. If the number is 1, then the program was designed to run in a ZCPR3 system with a permanent operating system memory buffer containing the environment descriptor. The program only has to store the address of that descriptor, and it can then adapt to any system in which it is run.

The environment or ENV address is stored in the two bytes immediately following the program type byte. Prior to ZCPR version 3.3, the address had to be installed into programs using a utility like Z3INS before they could be used. Starting with ZCPR33, the command processor installs the value as part of the process of loading the file from disk.

With ZCPR33, the type-3 program was introduced. These programs are not limited to execution at 100H, as all previous programs had been. The two bytes after the ENV address contain the address at which the code is designed to run. The Z33 command processor examines the program type byte, and if it is 3, it reads the load address from the header and proceeds to load the program to the designated address and execute it there.

The type-3 program made it possible to load and run programs at addresses other than 100H, but the address at which any given program file would run was still

fixed. In his column in the last issue, Bridger Mitchell described a fascinating and remarkable program structure that allows a program to run at whatever address it is loaded to. That same idea is the basis for the new type-4 program. Bridger's ANYWHERE program could be loaded manually to any address and then executed. Type-4 programs are loaded automatically by the command processor to the highest address in memory at which they can run without overwriting the command processor or any resident system extension (RSX) that is present. I would like to provide some additional details on how type-4 programs work and how they are constructed.

As with ANYWHERE, type-4 programs are derived from so-called (and, as Bridger pointed out, mis-named) page-relocatable or PRL files. Bridger defined and described those files in his column in the last issue, but another shot at it probably won't hurt. I will approach the subject somewhat differently—with a concrete example. Consider the short and simple program in Listing 1. It is set up for a starting address of 100H. Figure 1 shows the binary image of the sort one would see if the program were loaded with a debugger (e.g., DDT) and displayed.

If we change the argument of the ORG directive from 100H to 200H and assemble again, we get the results shown in Listing 2 and Figure 2. You should examine those results and note the things that have stayed the same and the things that have changed. Note in particular that only three bytes in the code have actually changed. One is the high order byte of the address of the initial jump instruction. The destination of that jump is in the program, and, since the program has moved up by 100H, the jump address has increased by an identical amount. The second change is in the data word containing the entry point address. Obviously that address changes when the program origin is changed. The third change is in the value loaded into the DE register pair. It is the address of the message string, which is likewise a part of the program.

Note that the argument of the jump to DOS has not changed. It is an absolute address outside the program. Therefore, it does not change.

Now let's look at a PRL file for the same program. I am not aware of any assemblers that can produce a PRL file directly. The usual procedure is to remove the ORG statement from the source code, assemble the program to a REL (normal relocatable format), and then use a linker to generate the PRL file from the REL file. Figure 3 shows the binary image of a PRL file produced using the SLR virtual-memory linker SLRNK+. Unfortunately, inexpensive linkers, such as SLRNK and ZLINK, are not able to produce PRL

```
       00 01 02 03     04 05 06 07     08 09 0A 0B     0C 0D 0E 0F
-------------------------------------------------------------------
0100   C3 13 01 5A     33 45 4E 56     03 00 00 00     01 48 65 6C
0110   6C 6F 24 0E     09 11 0D 01     C3 05 00
```

Figure 1. Binary image of the sample program in Listing 1.

```
       00 01 02 03     04 05 06 07     08 09 0A 0B     0C 0D 0E 0F
-------------------------------------------------------------------
0100   C3 13 02 5A     33 45 4E 56     03 00 00 00     02 48 65 6C
0110   6C 6F 24 0E     09 11 0D 02     C3 05 00
```

Figure 2. Binary image of the sample program in Listing 1 when linked to run at a starting address of 200H and loaded at 100H.

```
Z80ASM SuperFast Relocating Macro Assembler     Z80ASM 1.31 Page   1
PRLTEST Z80

    1         0100              org     100h
    2
    3 0100                entry:
    4 0100  C3 0113               jp      start
    5
    6 0103  5A 33 45 4E           db      'Z3ENV'
    7 0108  03                    db      3
    8 0109  0000                  dw      0
    9 010B  0100                  dw      entry
   10
   11 010D  48 65 6C 6C msg:      db      'Hello','$'
   12
   13 0113                start:
   14 0113  0E 09                ld      c,9
   15 0115  11 010D              ld      de,msg
   16 0118  C3 0005              jp      0005h
   17
   18                            end
 0 Error(s) Detected.
 27 Absolute Bytes. 3 Symbols Detected.
```

Listing 1. Simple example program assembled for a load address of 100H.

```
Z80ASM SuperFast Relocating Macro Assembler     Z80ASM 1.31 Page   1
PRLTEST Z80

    1         0200              org     200h
    2
    3 0200                entry:
    4 0200  C3 0213               jp      start
    5
    6 0203  5A 33 45 4E           db      'Z3ENV'
    7 0208  03                    db      3
    8 0209  0000                  dw      0
    9 020B  0200                  dw      entry
   10
   11 020D  48 65 6C 6C msg:      db      'Hello','$'
   12
   13 0213                start:
   14 0213  0E 09                ld      c,9
   15 0215  11 020D              ld      de,msg
   16 0218  C3 0005              jp      0005h
   17
   18                            end
 0 Error(s) Detected.
 27 Absolute Bytes. 3 Symbols Detected.
```

Listing 2. Simple example program assembled for a load address of 200H.

```
=========================================================================

        00 01 02 03     04 05 06 07     08 09 0A 0B     0C 0D 0E 0F
-------------------------------------------------------------------------
0100    00 1B 00 00     00 00 00 00     00 00 00 00     00 00 00 00
0110    00 00 00 00     00 00 00 00     00 00 00 00     00 00 00 00

0200    C3 13 01 5A     33 45 4E 56     03 00 00 00     01 48 65 6C
0210    6C 6F 24 0E     09 11 0D 01     C3 05 00 20     08 01 00

=========================================================================
```

Figure 3. Binary image of PRL file produced for the same test program and loaded at address 100H. Some memory regions containing bytes of 00 have been omitted from the display here.

files. Later we will show you a method, though somewhat tedious, that allows you to construct a reasonable approximation to a PRL file using an ordinary assembler (no linker at all).

SLRNK+ actually cannot produce a PRL directly using the source code as listed. The SLR manual discusses in a somewhat opaque way the technique for generating a correct PRL file. The problem is that the one-page nearly empty header at the beginning of the program is not generated. Joe Wright invented the trick of linking in the file SLR.REL derived by assembling source code with the sole statement

```
DS      256
```

This allocates one page of memory. The PRL file is produced by the linker command

```
SLRNK+ TEST/K,SLR,TEST,/E
```

The term "TEST/K" defines the output file, the term "SLR" allocates the empty header, and the term "TEST" links in the actual program code.

You should notice in Figure 3 the following things. First, the PRL file begins with a one-page header, which is entirely zero except for a word at address 101H (you can't tell from this example that it is a word, but it is). This word is the length of the code, 001BH or 27 decimal in this example. The program code itself begins on the next page (200H) and is the same as the code in Figure 1.

The other new bytes in the PRL file are those that follow the last byte of the program code. These bytes comprise the relocation bitmap that Bridger Mitchell described in his column in the previous issue of TCJ. The first byte is 20H, which expanded to binary is 00100000. This means that the third byte in the program code is the high byte of an address that must be relocated to make the program code execute at an address other than 100H. Indeed, the third byte is the address to which the JP instruction jumps. The second byte in the bitmap is 08H or 00001000 binary. This tells us that the 13th byte in the program code is an address that has to be relocated when the program is relocated. Indeed, this is the address of the start of the program in the Z-header. The third byte in the bitmap is 01H or 00000001 binary. It tells us that byte 23 is an address. If we look carefully, this is the address part of the "LD DE,MSG" instruction.

How Does ZCPR34 Load and Execute a Type-4

Bridger Mitchell explained last time in some detail how a PRL file can be relocated to run at any address. It really is

not necessary to understand all the details. The basic idea is that the bitmap tells a loader which bytes in the code to adjust.

The Z34 command processor has a special mechanism for processing type-4 programs. After the command processor has located a transient program, it loads the first record of the file into the default buffer at 80h. Here it can examine it to see what kind of program it is. If it is a standard CP/M program or a type-I or type-2 Z-System program, it sets up the load address as 100H and proceeds to load the entire file into memory, starting over with the first record. If it is a type-3 program, the same procedure is followed except that the load address is extracted from the program header.

With the type-4 program things are not so simple, because the load address has to be calculated and the code has to be relocated. Z34 gets these tasks accomplished in a very clever and tricky way. It could have done all the work itself, but that would have added a lot of code to the command processor. Instead, we took advantage of the fact that a PRL file has that two-record header with almost nothing in it. To make a type-4 program, we overlay onto this header a special loader program. Z34 executes the code there to calculate the load address and then to perform the code relocation.

The loader is available in HEX format (TYP4LDR.HEX) and can be grafted onto the PRL file using the command

```
MLOAD file=file.PRL,TYP4LDR
```

where 'file' is the name of the PRL file that you want to convert to a type-4 executable file.

By putting the loader code in the program rather than in the command processor, we provide additional flexibility. TYP4LDR calculates the highest address in the TPA to which a program can be loaded, but other loaders could return the address of the RCP or FCP and make possible self-installing modules. Clever users will undoubtedly come up with some other interesting applications that use special header code.

How Do We Make a PRL File

The easiest way to make a PRL file, and from that a type-4 program, is with a capable linker like LINK from Digital Research or SLRNK+ from SLR Systems. LINK came with my CP/M-Plus computer; I do not know how much it costs or how to obtain it otherwise. SLRNK+, which offers many very useful and powerful features besides the ability to make PRL files, costs $195. For someone who wants to experiment casually with

```
IMAKEPRL.HEX<cr>          ; Ready to load PRL maker routine
R,<cr>                    ; Load it to address 100h
ITEST100.COM<cr>          ; Ready to load program ORGed for 100h
R100<cr>                  ; Load it to address 200h (offset 100)
ITEST200.COM<cr>          ; Ready to load program ORGed for 200h
R<nextaddr-100><cr>        ; Load it to proper offset
s10E<cr>                  ; Ready to patch in code size
low-nextaddr<cr>          ; Low byte of code size
high-nextaddr<cr>         ; High byte of code size
.<cr>                     ; End patch with period
G<cr>                     ; Run MAKEPRL code at 100h
X<cr>                     ; Display registers -- note value of HL
F103,1FF,0               ; Clean up the header area
G0<cr>                    ; Exit from DDT
```

Figure 4. Commands issued to DDT to produce a PRL file from two COM files assembled to run at addresses of 100h and 200h.

type-4 programming, this is probably too much money to spend. If you are not going to do it very often and don't mind a little work, you can hand craft a PRL file using a debugger like DDT. I will take you through the procedure using our sample program from Listing 1.

Making the bitmap is the hard part of the procedure. You should key in the program called MAKEPRL.Z80 in Listing 3 and assemble it to a HEX file. We will use that code in the debugger first to make a "byte-map" and then to convert the byte-map into a bitmap. We assume that we have already assembled versions of the program with ORGs of 100H and 200H.

To construct the PRL file, we invoke the debugger (assumed to be DDT) and issue the commands shown in Figure 4. The first pair of commands loads the utility program MAKEPRL. The next two lines load the version of our program that was assembled to run at 100H. At this point we have to note the "next load address" reported by the debugger (I suggest you write it down). Now we load the version of the program assembled to run at 200H so that it follows right on the end of the 100H version. To do this, we use an offset value in the "R" command that is 100H lower than the "next address" that was reported a moment ago.

There is one other very important step we need to perform at this point. MAKEPRL has to be told the address at which the second program image was loaded. The value is patched in at address 10EH using the commands shown in Figure 4 starting with "S10E". For our example program, the next address is reported as 0280. Therefore, low-nextaddr is 80 and the high-nextaddris 02.

Now we let MAKEPRL do the hard part by running it with the "G" command. When it is finished, we need to examine the value in the HL register, since it tells us the next address after the bitmap. After leaving DDT we have to save the code image from 100H up to but not including that address. For the example program, the value in HL is reported to be 290H. Since we are presumably running Z34 and have the type-4 SAVE program, we save the result using the command

```
SAVE 100-28F PRLTEST.COM
```

If you do not have the type-4 SAVE,you will have to calculate the number of sectors to save.

Figure 4 lists one DDT command that we did not discuss. The "F103,1FF,0" command fills the part of the header after the code size word with zeros. This makes the file look prettier, but it is not absolutely necessary, especially if you are later going to overlay the type-4 loader as described below.

Listing 3. Utility program to perform the hard part of making a PRL file using a debugger.

```
; This code, which assists in the generation of a PRL file from a
; pair of COM files assembled for execution at 100H and 200H, is by
; no means optimized for speed or size. I have tried to optimize it
; for clarity!

        org     100h

        db      0               ; Standard PRL header (and NOP)
size:   dw      0               ; PRL file size (filled in by code)

        jp      start

        db      'CODE200:'      ; Identification string
c200:
        dw      0               ; Patch to address of code linked to 200h

start:

; The first step is to compute the size of the code and store the
; value at address 101h as required for a PRL file.  We also put
; this value in BC. We set up DE to point to the code assembled
; for 200H and HL to point to the code assembled for 100H.

        ld      hl,(c200)       ; Start of code for 200h
        ld      de,200h         ; Start of code for 100h
        xor     a
        sbc     hl,de           ; Difference is assumed size of code
        ld      (size),hl       ; Store in proper place for PRL file
        ld      b,h             ; ..and in BC
        ld      c,l

        ld      hl,(c200)
        ex      de,hl           ; DE -> code for 200h, HL -> code for 100h

; Now we subtract the code for 100h from the code for 200h to
; generate the map of bytes that are addresses that have to be
; relocated.  There will be a byte of 01 corresponding to each byte
; in the code that is the high order byte of an address that must
; be relocated. There will be bytes of 00 everywhere else.

bytemap:
        ld      a,(de)          ; Get byte from 200h version
        sub     (hl)            ; Subtract byte from 100h version
        ld      (de),a          ; Replace 200h code with byte map
        inc     hl              ; Point to next bytes
        inc     de
        dec     bc              ; Any more to do?
        ld      a,b
        or      c
        jr      nz,bytemap      ; Loop until done

; Now we have to compress the byte map into a bit map, taking
; each 8 bytes of the byte map and packing the values into a
; single byte in the bit map. The result is written immediately
; following the code (i.e., at the location of the code linked
; to 200h).

        ld      hl,(size)       ; Get number of bytes in byte map
        ld      b,3             ; Divide by 8 (2 to the 3rd power)
divide:
        xor     a               ; Clear carry
        rr      h               ; Rotate H right, low bit to carry
        rr      l               ; Rotate L right, carry into high bit
        djnz    divide          ; Repeat 3 times
        ld      (mapsize),hl    ; Save the value

        ld      de,(c200)       ; Point to byte map
        ld      hl,(c200)       ; Point to bit map (same place!)

makemap:
        ld      b,8             ; Process block of 8 bytes into 1 byte
        ld      c,0             ; Initial value for byte in bit map
```

```
makebyte:
        ld      a,(de)          ; Get byte from byte map
        inc     de              ; Advance pointer
        rr      a               ; Move relocation bit into carry flag
        rl      c               ; Move carry flag into byte in C
        djnz    makebyte        ; Repeat 8 times

        ld      (hl),c          ; Put result into bit map
        inc     hl              ; ..and advance its pointer

        push    hl
        ld      hl,(mapsize)    ; See if we are done
        dec     hl
        ld      (mapsize),hl
        ld      a,h
        or      l
        pop     hl
        jr      nz,makemap

        rst     38h             ; Breakpoint to end program

mapsize:
        ds      2               ; Scratch area

        end
```

The PRL files made this way can be used to make type-4 programs, and they can be used in Bridger Mitchell's ANYWHERE program. However, we should point out that these PRL files are not as efficient as those produced by a linker. We assumed that the code in the COM files extended to the end of the last record in the file.

Perhaps you can build on my simple method and figure out how to extend it to produce an optimal PRL file just like the one from SLRNK+. It would also not be too difficult to write a program using routines in SYSLIB to read in the pair of COM files and generate a PRL file from them completely automatically. The most elegant method for doing this would use random-record writes. I invite readers to send me such a program. ■

Database Primer

(Continued from page 11)

I would set up the church system using a data entry set for additions as a separate batch file which is appended to the main set after saving a copy of the main set before the additions, along with the batch file. Some systems can trash the files if the system goes down (or is turned off) with a file open, so we want to minimize the time that the main file is open. Keeping the main file open should not be an operator option. In order to catch duplicate entries, we could open the main file for read only, search for a dup, close the file, and either advise of the dup or write the record to the batch entry file. For changes or deletions I would follow the same scheme, reading the main file to verify that the record does exist, then writing the record to the update batch file. After all the data entry is completed, I would add the new entries, replace the changed records, and remove the deleted records. I also feel that the main data file before the changes and the batch files should be saved to a floppy for archiving.

The above batch operation is OK for the single user church application, but not for most multiuser POS, inventory, or accounting systems. For example, an airline reservation system has to update the data in real time so that the next travel agent can determine the available seats on a specific flight. Even though batch operations are not used, it is still necessary to write transaction files recording who changed what when, so that the files can be rebuilt or in order to determine who made certain changes.

Next Time

In the next issue, we will continue our discussion of the factors which need to be considered in order to select the most suitable data base sytems. We start on the next issue while this issue is still at the printer, so don't delay in sending your letters, comments, or articles. ■

New Microcontrollers Have Smarts

Program with ROM based On-Chip BASIC or Forth

by R.E. McCain

For many years the industrial and embedded controller market has been dominated by machine dependant assembly language programming, primarily because of the cost involved in creating an environment capable of supporting a higher level language. In order to use a high level language it was often necessary to add floppy disks and controllers, or write your own language and load in EPROM. Both the Apple][and IBM/PC had ROM BASIC of this sort, but the user was forced to use external program storage or use another system to develop and burn EPROM for application code.

Recently both Intel and Zilog have begun to supply microcontrollers with onboard ROM that may optionally be preprogrammed with either a BASIC or Forth operating system. The Intel 8052AH-BASIC chip with 8K of ROM costs about $25/unit for a floating point BASIC with many system features. Zilog's Z8671 chip has a 2K ROM with integer BASIC and costs about $8/unit. Zilog also plans to bring out a similar chip with Forth in ROM. Both chips have a serial I/O port, counter/timers, interrupts and a small internal memory. The Intel chip also requires a minimum of 1K of external memory while the Zilog chip may run in a standalone mode. On the 8052, three of the four I/O ports have been frozen into address/data and other system functions. The user is free to configure all four of the I/O ports on the Z8671.

Either of these chips allow the developer to use the BASIC as either a debug for assembly programs or as a fully functional BASIC interpreter. Because of the low cost, the Zilog part could easily be incorporated into many low end control applications, while the more expensive Intel unit is better suited for those applications requiring floating point operations or for multi-chip peripheral control uses. The 8052 has many development system capabilities not found in the Z8671, such as the ability to print numbers in hex or decimal, user defined console drivers, string manipulation, and the ability to program either EEPROM or EPROM. (Though he added many features and dedicated chips, Steve Ciarcia used an 8052 in his standalone Serial Eprom Programmer.)

I have been using an 8052 system for the past few months for development of an industrial controller. We used TASM-51 (available from PC-SIG, disk # 643) to assemble the console drivers and assembly subroutines and Procomm (PC-SIG disk # 499) to communicate with our PC/XT host system. Intel has done an outstanding job of providing a road map through the 8052 with their *MCS BASIC-52 User's Manual*. It actually provides real world hints and tips that make using the system enjoyable!

An 8052 based system is shown in Figure 1. Note that this could be the schematic both for the development prototype (using the 8052AH-BASIC chip) and the final product (using an 80321 chip) simply by changing the jumper on pin 31 of U1. 8051 family chips come in many different flavors and enhancements, but most of the development effort for even the most exotic could be accomplished on a stand-alone system using your personal computer as the host terminal.

Let's take a brief look at the features of this system. It can interface to any keyboard or switch matrix. A 16 character or numeric display can be driven from U8 and a bus structured LCD provides user feedback from the system. There are 6 byte wide I/O ports which may be configured as inputs, outputs or bidirectional. The PWM command in BASIC

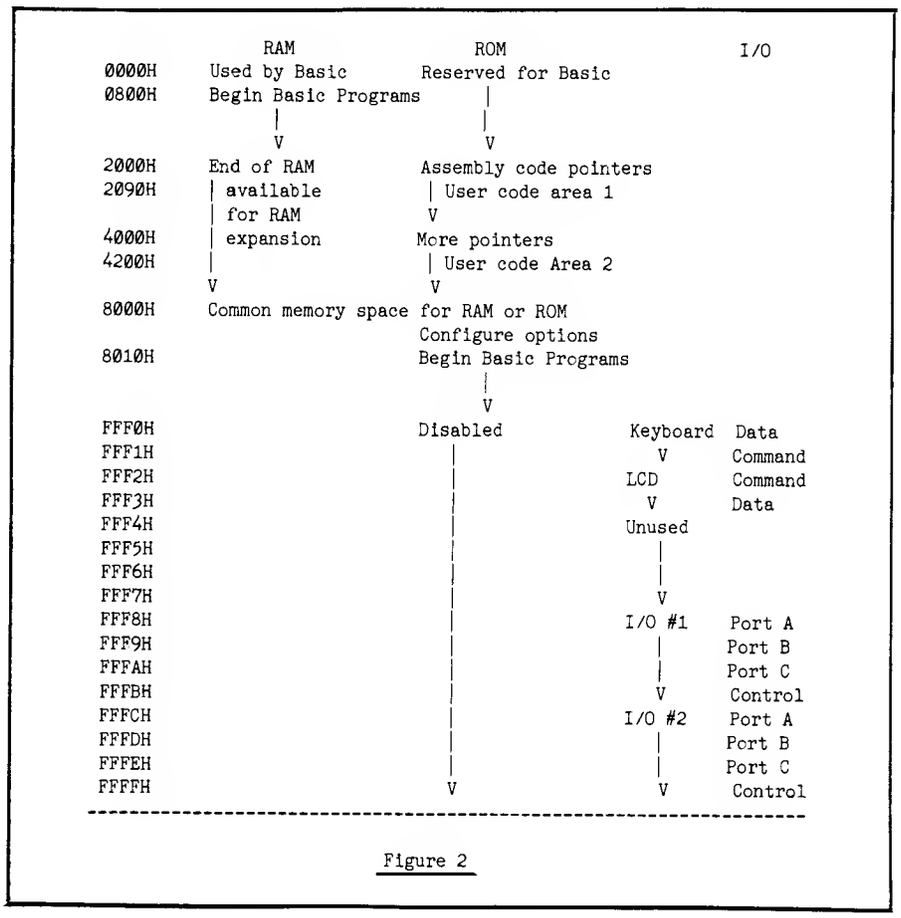
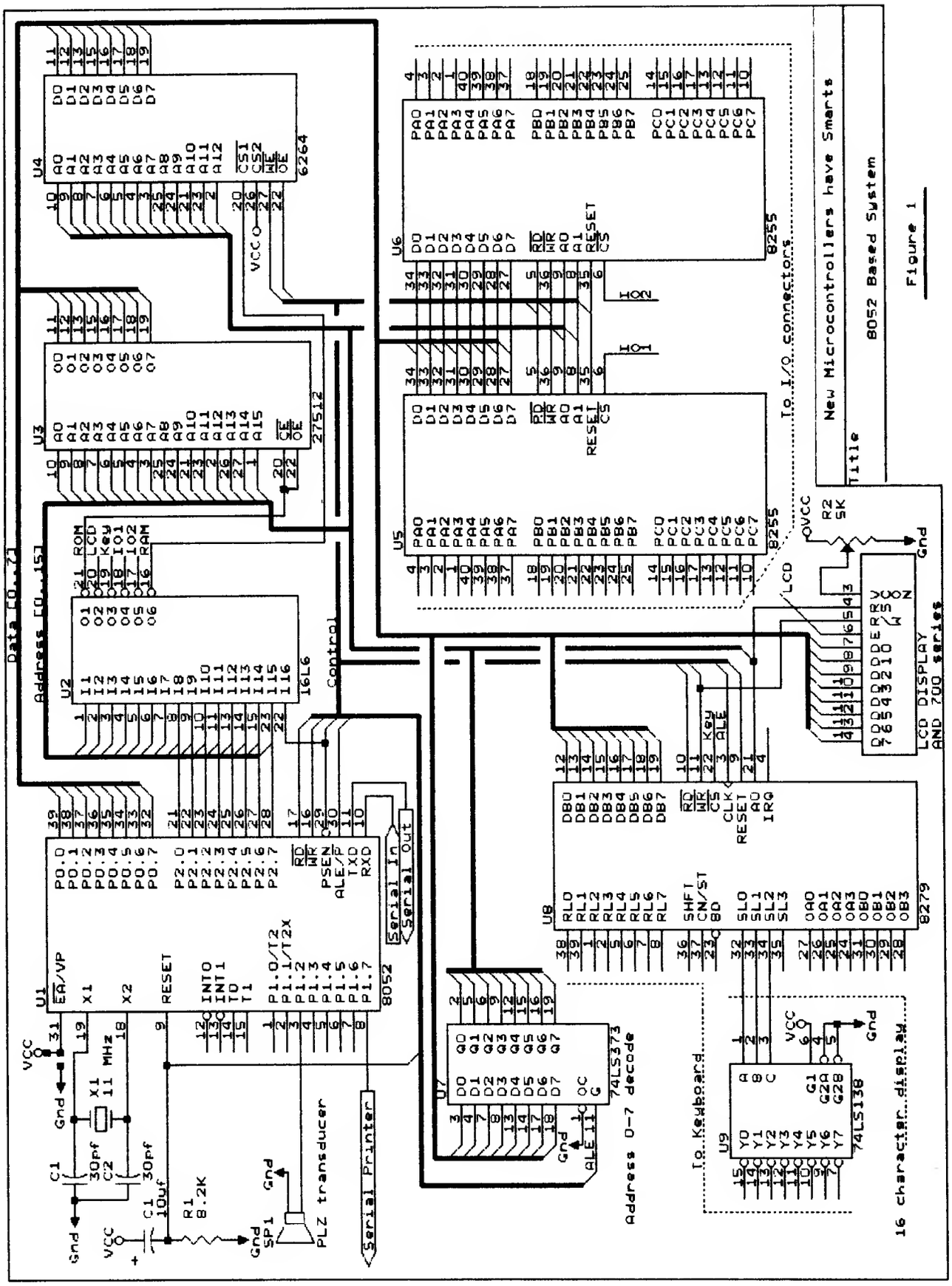


Figure 2


New Microcontrollers have Smarts
8052 Based System
Figure 1
Data [0...7]
Address [0...15]
Control
I/O connectors
LCD DISPLAY
AND 700 series
Serial In
Serial Out
Serial Printer
PLZ transducer
Address 0-7 decode
16 character display
To Keyboard
8052
8255
8279
74LS373
74LS138
6264
27512
16L6

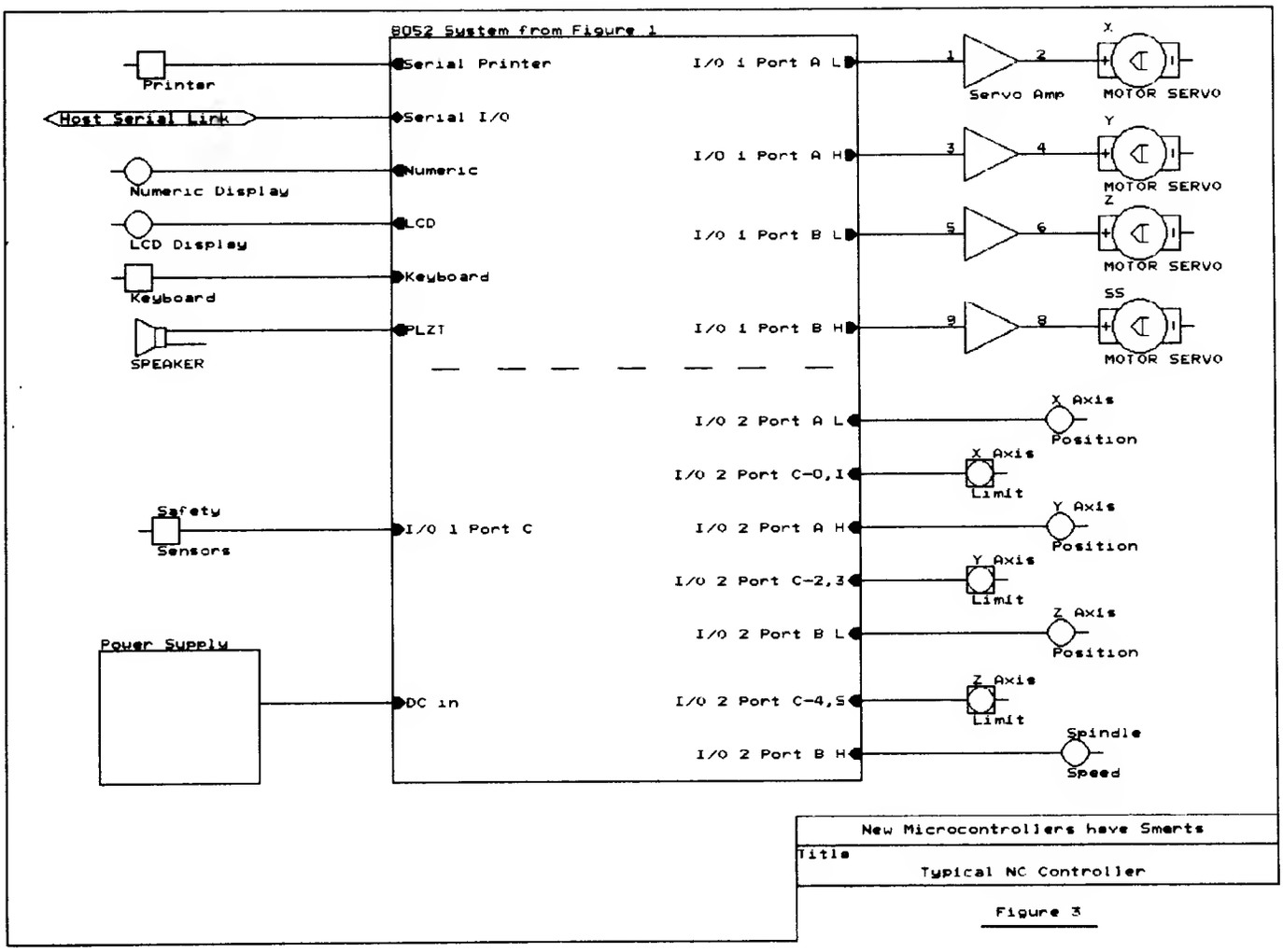


Figure 3

can drive the PLZ transducer with a variety of sounds.

There is 8K of RAM (of which BASIC uses 512 bytes) and 64K of EPROM (0000H to 1FFFFH is reserved for BASIC). In an 80321, the space from 0000H to 1FFFH could be loaded with either the BASIC operating system or used for custom code. (The source code for the BASIC is available through INSITE—the Intel user group). Assembly code usually occupies the space from 2000H to 7FFFH where the driver pointers are mapped, and the space from 8000H to FFEFH is used for tokenized BASIC program storage. The default addressing for the RROM# commands begins at 8010H. You may load as many different basic programs into EPROM as you need (until you run out of memory) and access them by number or use the first program as a menu to point to the others. The space from FFF0H to FFFFH is used for the memory mapped I/O devices, which are accessed through the PAL at U2. See Figure 2 for the memory map. The 8052 has a memory mapping scheme which allows you to use up to 96K of memory by switching from RAM to ROM in the space below 8000H.

The system could easily be used for an NC (Numeric Control) application such as that in Figure 3, where it could control a machine tool. In such an application, you may want to increase the RAM to 32K if large downloads are anticipated. An advantage of such a system is that one could have a battery backup for power failures or transient protection.

Our NC controller uses all the features of this system. The serial I/O is used to download and save setups. The local keyboard and motion switches allow the operator to control and interact with the system. The LCD provides operator feedback, while the numeric display gives relative position of the work in the X,Y and Z axes. The PLZ transducer generates warning and error alerts. The I/O controllers are used to control the servo motors, to sense position and end of travel, and to monitor any miscellaneous safety functions, such as operator hazards, low fluid levels, etc.

A real world NC controller might require more precision than our example allows, and a third I/O controller could be mapped to the unused addresses from FFF3H to FFF7H to allow the position sensors 12 bit accuracy. We used servo motors because of the high slew rates possible, but some applications might use stepping motors to improve positioning accuracy. The real world application might also require a multiple redundant approach for failsafe operation. It would be possible to design an inexpensive multiprocessor system with coincidence checking and other high reliability features to minimize downtime.

The flexibility of the floating point BASIC comes into play with the NC controller, as we can access the floating point routines from an assembly level, and the controller could easily accept and send setups in the various formats used in the industry. ■

you see here.

THE AMPRO LITTLE BOARD™/286

Big power for smaller systems.
Little Board/286 is the newest member of our family of MS-DOS compatible Single Board Systems. It gives you the power of an AT in the cubic inches of a half height 5 1/4" disk drive. It requires no backplane. It's a complete AT-compatible system that's functionally equivalent to the 5-board system above. But, in less than 6% of the volume. It runs all AT software. And its low-power requirement means high reliability and great performance in harsh environments.

Ideal for embedded & dedicated applications. The low power and tiny form factor of Little Board/286 are perfect for embedded microcomputer applications: data acquisition, controllers, portable instruments, telecommunications, diskless workstations, POS terminals . . . virtually anywhere that small size and complete AT hardware and software compatibility are an advantage.

Compare features.
Both systems offer:
- 8 or 12MHz versions
- 512K or 1Mbyte on-board DRAM
- 80287 math co-processor option
- Full set of AT-compatible controllers
- 2 RS232C ports
- Parallel printer port
- Floppy disk controller
- EGA/CGA/Hercules/MDA video options
- AT-compatible bus expansion
- A wide range of expansion options
- IBM-compatible Award ROM BIOS

But only Little Board/286 offers:
- 5.75" x 8" form factor
- EGA/CGA/Hercules/MDA on a daughterboard with no increase in volume
- SCSI bus support for a wide variety of devices: Hard disk to bubble drives
- On-board 1Kbit serial EPROM. 512 bits available for OEMs
- Two byte-wide sockets for EPROM/RAM/NOVRAM expansion (usable as on-board solid-state disk)
- Single voltage operation (+5 VDC only)
- Less than 10W power consumption
- 0-70°C operating range

*AT is a Registered Trademark of IBM Corp

Better answers for OEMs.
Little Board/286 is not only a smaller answer, it's a better answer . . . offering the packaging flexibility, reliability, low power consumption and I/O capabilities OEMs need . . . at a very attractive price. And like all Ampro Little Board products, Little Board/286 is available through representatives nationwide, and worldwide. For more information and the name of your nearest Rep, call us today at the number below. Or, write for Ampro Little Board/286 product literature.

408-734-2800
Fax: 408-734-2939 TLX: 4940302

AMPRO
COMPUTERS, INCORPORATED
1130 Mountain View/Alviso Road
Sunnyvale, CA 94089

Reps: **Australia**-61 3 720-3298; **Belgium**-32 87 46.90.12; **Canada**-(604) 438-0028; **Denmark**-45 3 66 20 20; **Finland**-358 0 585-322; **France**-331 4502-1800; **Germany, West**-49 89 611-6151; **Israel**-972-3 49-16-95; **Italy**-39 6 811-9406; **Japan**-81 3 257-2630; **Spain**-34 3 204-2099; **Sweden**-46 88 55-00-65; **Switzerland**-41 1 740-41-05; **United Kingdom**-44 2 964-35511; **USA**, contact AMPRO.

Advanced CP/M
Extending the Operating System

by Bridger Mitchell

Sooner or later, the CP/M user bumps up hard against the limitations of the operating system and wonders—can something be done? Yes, CP/M can be enhanced at several levels.

Command Processor Extensions

A great deal of effort has been directed to improving the external, command-processing part of CP/M® . The "command shell" is readily replaced; it is the most immediately noticed component; and it can be extended by placing added code in files that can be manipulated by the existing system. Z3PLUS, NZ-COM, and ZCPR34 are the latest achievements at this level; they both replace the command processor and provide well-defined resident buffers for communication between successive tasks. There have been a number of other replacement command processors, such as CNIX® , CONIX® , and QPM® .

BDOS and BIOS Extensions

The next level of extension is adding new operating system functions—extra BDOS calls. CP/M Plus® provides a particularly convenient method, called a Resident System Extension (RSX), for adding such capabilities. Under CP/M Plus, an RSX can be attached to a program that needs extended services and loaded automatically along with the original program. It is rather easy to modify existing BDOS functions, for example, to keep statistics on the frequency with which key files are accessed.

More elaborate RSXs have been developed for CP/M Plus to emulate the CP/M 2.2 BIOS file functions so that programs that make direct BIOS calls can run under CP/M Plus by attaching the necessary RSX but leaving the program itself unchanged. The most complex CP/M Plus RSX is the brand-new Z3PLUS system. Within the single RSX are the ZCPR34® command processor, and program and RSX loader, the Z-System buffers for named directories, resident commands, messages, command line, file control block, and environment descriptor.

Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); BackGrounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.

Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, and via Z-Node #2, (213)-670-9465.

However, more basic changes to CP/M 2.2, such as adding time and datestamping or redirection of console and printer services, are much more difficult. They are necessarily inner extensions to CP/M and require intimate knowledge of the BDOS and BIOS.

Another area that frequently needs extension is an interface to the hardware. The original BIOS may provide for only a limited number of disk formats. Function keys that transmit escape sequences may need to be timed and mapped into defined character strings. Printers could need character translation tables to generate control sequences or foreign characters. These system extensions are primarily BIOS modifications.

The most extensive extension of CP/M 2.2, and the most complex piece of software I have ever written, is BackGrounder ii. It is installed as an RSX that makes extensive modifications to selected BIOS and BDOS functions, replaces the command processor, and adds virtual swapping memory to the basic system. The result is effectively *two* virtual CP/M computers on one machine, with user-controlled ability to switch between two running programs at any time, and to preserve the exact screen display of each on most terminals.

RSXs for CP/M 2.2

An RSX is not necessarily the first, or the best, choice for adding capability to a computer system. The operating system can be extended by rewriting the BIOS (provided the source code is still available!). This is the most fundamental approach. Specific to one computer, it is permanent and totally non-portable.

The system can be extended for only a single application, as when a program is run by first loading a debugger. Quick and dirty, this approach is sometimes useful for temporarily modifying system behavior or pre-testing a more permanent approach.

A resident system extension is a versatile, intermediate approach. An RSX can be installed and left in place to extend the system for as long as it is needed, then removed to restore the original system and full memory. If well designed, it can be quite portable. It can be used to add both BDOS and BIOS features to the vanilla CP/M 2.2 system. And several RSXs can be used together to combine system power.

In this issue's column I discuss System Extensions that modify the behavior of the BDOS and/or BIOS. Because these services must essentially be available at all times (whenever a BDOS or BIOS call is made), the extension code must be __resident__. This is unlike extensions of the command processor, which can be reloaded from the system tracks (or from a file) when a new command is ready for processing.

I will limit the examples to extensions for the CP/M 2.2 operating system (and its clones). CP/M Plus already includes RSX capability for BDOS (but not BIOS) extensions and it also includes provision

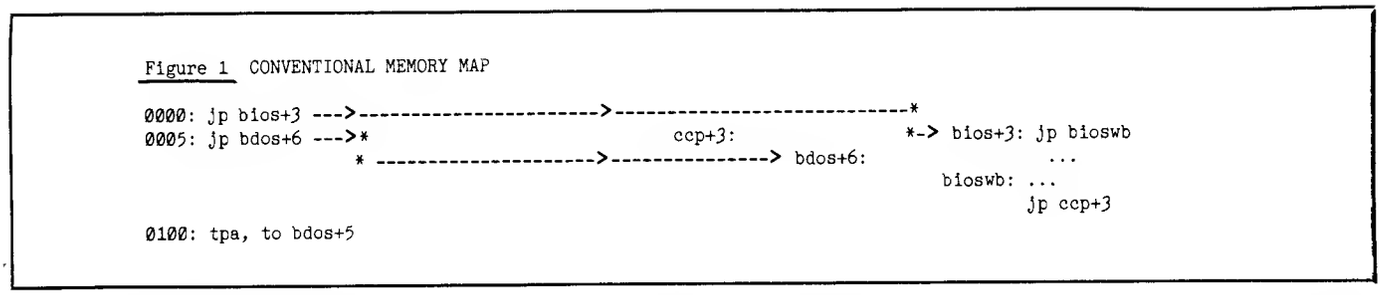


Figure 1 CONVENTIONAL MEMORY MAP

for character-device redirection (so that, for example, the printer output can be diverted into a file). The basic ideas, however, carry over to CP/M Plus. Indeed, I have used them effectively in implementing both Z3PLUS and DosDisk on CP/M Plus computers.

No Free Lunch

Although smart coding and clever uses of the RSX concept can greatly extend the vanilla CP/M system, there are limits imposed by a 64K address space Z-80 operating system. As I was preparing this column a potential DosDisk customer called, hoping that DosDisk would let him run Lotus 1-2-3 on his Z-80 computer with 8 inch disks! I trust that anyone who has read this far has somewhat more modest expectations.

The Conventional CP/M 2.2 System

In the remainder of this column I will describe how an RSX modifies the conventional memory map and flow of control, using a standard RSX header data structure. We will then turn to a complete example.

In order to see how RSXs fit into the CP/M 2.2 system, let's first review a few key points about a "vanilla" system.

The end of the TPA (transient program area), the memory available to application programs, is specified by the address stored on page-zero at location 0006. Initially, that value is the address of the entry to the BDOS, and a "call 5" instruction will jump directly to the BDOS. Beginning at the address whose value is stored at 0006, memory is said to be "protected"—it is not available to applications, and data and code in that area should remain "resident" from one application to the next.

The command processor is assembled or linked to be loaded in the top 2K of the TPA memory. The command processor is not resident, and that 2K of memory is available to applications. Therefore, on each warm-boot—a jump or call to 0000—the BIOS reads a fresh copy of the command processor into high memory. (It may also read in a fresh copy of the BDOS, depending on the implementation in the BIOS.)

Figure 1 shows the essential flow of control. The address at 0006 is "bdos + 6", the entry to the BDOS, and the TPA extends from 100h to bdos + 5. A jump to 0000 vectors to "bios + 3", which jumps to "bioswb", the BIOS warm-boot routine. That routine reloads the command processor, installs fresh values of "bios + 3" and "bdos + 6" at 0001 and 0006, and then jumps to the command processor entry "ccp + 3" to process the next command.

The Standard RSX Header

In order to insert a resident extension into this scheme we must deal with memory management, BDOS and BIOS calls, provide for removal of the RSX when it is no longer needed, and anticipate the loading of other RSXs while this one is active.

In addition to being intrinsically complex, implementing all of this is tedious and error-prone. Several years ago, to provide a standard approach, I established a Plu*Perfect Systems RSX header—a uniform data structure at the beginning of every RSX—to permit several RSXs to coexist peacefully. And for this column I extended JetLDR to recognize a relocatable RSX module in named-common ZRL relocatable format and load it automatically.

The header (Figure 2) begins with three jumps to routines in the RSX—to the BDOS intercept routine "rsxstart", to the warm-boot intercept routine "rsxwb", and to the removal routine "rsxremove".

Next come three addresses. First, "rsxwba" is the address of the (original) BIOS warmboot vector which is used to, hopefully, correct the problems created by programs that erroneously change the contents of 0001. The second word is the "protect" address, the lowest byte in the RSX that must be in protected memory. This is followed by the address of the null-terminated ASCII name of the RSX.

Two jumps and a final word complete the header. The chain of a warm-boot (jump 0000) flows through "rsxnext", which points to either the next-higher RSX, or to the command processor entry. Similarly, the call BDOS chain flows through "next", which points to either the next-higher RSX, or to the BDOS entry. The final item, "nextwb" holds the address from the BIOS warm-boot vector at the time the RSX is loaded, so that it can be restored by the remove routine.

Adding an RSX to Memory

In order to add an RSX to a running system, we must arrange to:

1) Allocate sufficient memory for the RSX.

2) Deduct that memory from the available TPA, so that an application will not attempt to use the memory occupied by the RSX.

3) Prevent the BIOS from undoing the memory allocation at the next warm-boot.

We will locate the RSX in high memory, immediately below the command processor—see Figure 3. To do this, we change the value at 0006 to point to the first byte of the RSX, and arrange to have a jump instruction there that (eventually) causes control to enter the BDOS (this is the initial jump in the standard RSX header). The TPA is now reduced to the area 100h to "rsx1-1".

To keep the BIOS routine "bioswb" from undoing our handiwork by resetting the value of 0006, we must either modify the "bioswb" routine, get control back

Figure 2. Standard RSX Header

```
rsx:    jp      rsxstart        ; BDOS intercept                ;  00
        jp      rsxwb           ; warm-boot intercept           ; +03
        jp      rsxremove       ; remove-rsx entry              ; +06
rsxwba: dw      $-$             ; original 0001 value           ; +09
rsxprot:dw      rsx             ; lowest rsx address            ; +0B
        dw      rsxname         ; -> rsx name                   ; +0D
rsxnext:jp      $-$             ; -> next wb or ccp entry       ; +0F
;
next:   jp      $-$             ; -> next rsx or bdos           ; +12
nextwb: dw      $-$             ; original bios+4 addr          ; +15
```

Figure 3 MEMORY MAP WITH ONE RSX

```
              *-->------------------------>----------------------->---*
0000: jp bios+3-*         *-----------------<----------------------<---+------------------<---*
0005: jp rsx1   ----------+> rsx1: jp entry1   *-> ccp+3:                *--> bios+3: jp rsx1+3-*
                          *--> +3: jp rsx1wb  /          *-> bdos+6:
0100: tpa, to rsx1-1                 ...       /          /
                                 +F: jp ccp+3-*          /
                               next1: jp bdos+6 ---------*
                              rsx1wb: ...
                                      jp rsx+F
                              entry1: ...
                                      jp next1
```

Figure 4 MEMORY MAP WITH TWO RSXs

```
              *-->------------------------>----------------------------------------->---*
0000: jp bios+3 *         *-----------------<-----------------------------------------<---+------------------<---*
0005: jp rsx2   ----------+> rsx2: jp entry2     rsx1: jp entry1   *-> ccp+3:              *--> bios+3: jp rsx2+3-*
                          *--> +3: jp rsx2wb     *->+3: jp rsx1wb  /          *-> bdos+6:
0100: tpa, to rsx2-1                 ...        /       ...       /          /
                                 +F: jp rsx1+3--*    +F: jp ccp+3-*          /
                               next2: jp rsx1       next1: jp bdos+6 ---------*
                              rsx2wb: ...          rsx1wb: ...
                                      jp rsx2+F             jp rsx1+F
                              entry2: ...          entry1: ...
                                      jp next2              jp next1
```

Here, the labels ''ccp'', ''bios'', and ''bdos'' refer to the base address of the corresponding CP/M operating system segment. The entry to the bdos is at bdos+6; don't confuse it with the common equate for the page-zero vector: ''bdos equ 0005''.

after that routine has completed, or prevent it from executing. Prevention is the only portable solution; and it must be done intelligently.

The System Loadstone

The "obvious" method of bypassing the warm-boot routine is to change the address at 0001 to point to a copy of the (modified) BIOS jump table located in the RSX. However, this would be fundamentally wrong, because this is the *one* address in the CP/M system that must remain fixed. Why? Consider what will happen when a second RSX is being loaded. It will be unable to locate the BIOS and therefore cannot correctly intercept BIOS functions!

The warm-boot address at 0001 should never be altered! It is the one fixed point, the lodestone, of the entire CP/M system. Instead, the warm-boot chain should be intercepted at the BIOS vector (at bios+4) and redirected to the RSX from that point.

Unfortunately, this absolutely essential role of 0001 has apparently not been widely understood. Both Digital Research (in XSUB)—the designers of CP/M!—and MicroPro (in the WordStar "R" command) occasionally commit the error of changing 0001, and their mistakes have been perpetuated by several public-domain programs as well.

Figure 3 shows a correctly modified warm-boot chain with the RSX installed. At 0000, control continues to jump to the BIOS, where it is *then* redirected to "rsx1+3" and then to the RSX's warm-boot routine. Eventually, control flows to the copy of the command processor that is already in memory, bypassing the bios warmboot routine and without reloading the command processor. We will cover the details of the RSX warmboot routine in the example below.

A Second RSX

Adding a second RSX is similar to loading the first one (see Figure 4) and involves splicing the new RSX into the BDOS and warmboot chains. Memory is allocated below the first RSX, 0006 is redirected to the head of rsx2 where BDOS calls are intercepted and eventually vectored to rsx1. The warm-boot intercept at bios+4 is redirected to rsx2+3, which eventually vectors to rsx1's warmboot routine.

Removing an RSX

The standard RSX header includes an entry that is called to remove the routine. It deallocates the RSX's memory and removes itself from the BDOS and warm-boot chains. It also removes its BIOS intercepts, if any. If there is more than one RSX in memory, the lowest one must be removed first, to ensure that the addresses (or data) that are restored are correct. Each RSX's remove routine first determines that it is, in fact, lowest before executing the rest of the removal code.

The standard header anticipates that most RSXs will be self-removing, provided that they are the bottom RSX when the remove entrypoint is called. However, some RSXs, such as DateStamper and DosDisk, make extensive modifications to the BIOS and BDOS. A good deal of additional code and data would have to be resident in the RSX for these RSX's to be self removing. Therefore, these RSXs are removed by their own customized loaders. If the remove entrypoint is called by any other program it will do nothing except to return the carry flag clear, indicating that the RSX has not been removed.

OKDRIVES, or, Who's On Line?

This is the third Advanced CP/M column, and by now some readers will have anticipated my penchant for turning to an actual application to illustrate the

DosDisk™ -- An MS-DOS Disk Emulator for CP/M

DosDisk, for CP/M 2.2 and CP/M Plus Z80 computers, allows CP/M programs to use files stored on an MS-DOS (PC-DOS) floppy disk *directly* -- without intervening translation or copying. You can *log into* the pc disk, *including subdirectories*. Regular CP/M programs can read, write, rename, create, delete, and change the attributes of MS-DOS files. The disk, with any modified files, can immediately be used on a pc.

Preconfigured Versions are available for:

all Kaypros with a TurboRom
all Kaypros with a KayPLUS rom and QP/M
Xerox 820-I with a Plus 2 rom and QP/M
Ampro Little Board
SB180 and SB180FX with XBIOS
Morrow MD3
Morrow MD11
Oneac ON!
Commodore C128 with CP/M 3 and 1571 drive

The resident system extension (RSX) version uses about 4.75K of main memory (plus 2K for the command processor). For the SB180 and SB180FX, a banked system extension (BSX) version is also available; it needs about 5K of the XBIOS system memory and *uses no main memory*.

A Kit Version requires *advanced assembly-language experience* in Z80 programming and *technical knowledge of your computer's BIOS*. You will need to write a special **DosDisk** overlay.

The BIOS must be able to be configured to use the physical parameters of an MS-DOS disk and to use the logical disk parameter header (dph) and disk parameter block (dpb) values supplied by **DosDisk**. The driver code itself (the code that programs the disk controller, reads and writes sectors, etc.) must reside *in the BIOS*.

On DateStamper, QP/M and CP/M 3 systems **DosDisk** automatically stamps MS-DOS files with the current date and time when they are created or modified.

DosDisk supports the most popular MS-DOS format: double-sided double-density 9-sector 40 track disks. It cannot format disks or run MS-DOS programs.

Z3PLUS™ --The Z-System for CP/M Plus

The *state-of-the-art* ZCPR version 3.4 system for CP/M Plus (CP/M 3) Z-80 computers *installs automatically* and retains CP/M Plus advantages -- fast disk operations, redirection of screen, keyboard and printer; automatic execution of submit files.

Z3PLUS is fully configurable and requires no assembly. It is shipped with key Z tools and will run most Z-System CP/M 2.2 utilities without modification.

==========

DosDisk and **Z3PLUS** are available directly from the author of DateStamper and BackGrounder ii:

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402

Name: _____

Address: _____

_____

Computer: _____

Operating system: _____

Disk format: _____

Check Product:

[ ] **DosDisk** preconfigured version ..... $ 30.00
[ ] **DosDisk** kit version ..... $ 45.00
[ ] **DosDisk** manual only ..... $ 5.00
[ ] **DosDisk** BSX and RSX,
for SB180/SB180FX with XBIOS ..... $ 35.00
[ ] **Z3PLUS** ..... $ 69.95
(in California, 6.5% sales tax) ..... _____

shipping/handling ..... $ 3.00

total enclosed ..... $_____

technical subject at hand. This example grew out of a conversation with Ben Grey, who was seeking a general-purpose method of limiting access to certain drives on a remote system.

Many users have computers with a mix of actual and "missing" logical drives. For example, A: and B: might be floppies and M: a ram disk.

How can an application program determine whether it should attempt to use a particular drive? Simply trying to select the drive with a BDOS call is playing Russian roulette. If the drive is invalid, or doesn't exist, the BDOS will complain with an error message and terminate the application with extreme prejudice—an abrupt warm-boot.

ZCPR34 allows the user to specify a vector of valid drives, stored in the Z-System extended external environment at offset 34h. The ZCPR34 command processor tests this vector before attempting to log in a drive, and the same test can be made by an application once it determines that it is indeed running in a system with an extended external environment. The extensions are listed in my previous Advanced CP/M column.

Frequently, however, the programmer wants his application to run smoothly on other CP/M systems that aren't (yet) running ZCPR34. And users occasionally need a method of taking an erratic drive "off line" for maintenance. Is there a general, portable method of (1) determining which drives are currently available? and (2) making individual drives inaccessible?

OKDRIVES in Figure 5 is one method of dealing with this challenge. It is a small RSX that maintains a vector of valid drives and monitors all BIOS disk-select calls. If a drive is selected that is not in the vector, it returns a select error. Otherwise, it allows the BIOS to select the drive. In order to set and change the list of valid drives, the RSX adds one BDOS function to the system. That function call serves the dual purpose of setting the valid (and invalid) drives, and reporting what the current setting is.

Structure of the RSX

OKDRIVES is written to be assembled into a ZRL (Z-ReLocatable) file and loaded with JetLDR, which I also described in the previous column.

The RSX is made up of the standard RSX header, custom code to perform the operations on the valid-drives vector, and an initialization section. I have written the RSX in a quasi-modular fashion, so that almost all of the tedious code to install and remove the RSX can be copied in a few blocks and reused in any other RSX.

This RSX uses one extended BDOS function number—241. When the RSX has been installed and the BDOS is called

Figure 5

```
title   okdrives.asm 6/25/88    (c) 1988 Bridger Mitchell
;
;
; This rsx sets the vector of valid drives allowed by the bios.
; If called with de == 0, it returns the current valid-drives vector.
;
; usage to set valid drives:
;
;       ld      c,DRIVEFN
;       ld      de,<vector>     ; bit 0 = A:, ..., bit 15 = P:
;       call    5
;
; usage to determine currently-valid drives:
;
;       ld      c,DRIVEFN
;       ld      de,0000
;       call    5
;
;
; We need an extended bdos function number.
;
DRIVEFN equ     241     ; 0F1h
ABORT   equ     0ffh
;
;
; Name the REL image with ''RSX'' plus 0-3 characters of identification.
; In this case, we've used the rsx's bdos function number (241).

name    RSX241          ;''RSX'' required

;
; All of the code within the bracketed regions is the same for any rsx
; loaded by JetLDR.  It can be copied intact when creating a different rsx.

;
; *---------- Plu*Perfect Systems RSX Extended Header----------------*
;/                                                                   \
;
; The rsx code goes in the CSEG (code segment).
;
CSEG

rsx:    jp      rsxstart                                        ;  00
        jp      rsxwb                                           ; +03
        jp      rsxremove                                       ; +06
rsxwba: dw      $-$                                             ; +09
rsxprot:dw      rsx                                             ; +0B
        dw      rsxname                                         ; +0D
rsxnext:jp      $-$             ; -> next wb or ccp entry       ; +0F
;
next:   jp      $-$             ; -> next rsx or bdos           ; +12
nextwb: dw      $-$                                             ; +15
;\                                                                   /
; *-------------------------------------------------------------------*
;
;
; The custom code for this rsx begins here.
;
;
rsxname:db      'OKDRIVES',0      ; nul-terminated name of rsx.
;
vector: dw      1111111111111111b ;  <-- set bits for valid drives
;               PONMLKJIHGFEDCBA  ;  <-- must be terminated by 'B' char.
;
;
; This RSX's bdos function.
;
; enter: c  =  DRIVEFN
;        de == 0 to get current ok-drives vector
;        de != 0 to set the current vector to de
; return:
;        hl = vector of ok drives
;
```

```
rsxstart:
        ld      a,c
        cp      DRIVEFN         ; if not our function
        jr      nz,next         ; .. on to the next rsx/bdos
        ld      a,e             ; set vector?
        or      d
        jr      nz,set          ; ..yes
get:    ld      hl,(vector)     ; no, return the drive vector in hl
        ld      a,l             ; return a != 0
        ret
;
set:    ex      de,hl
        set     0,l             ; ensure drive A: always valid
        ld      (vector),hl     ; save the new drive vector
        ld      a,l             ; and return it in hl
        ret

;
; The bios seldsk intercept
;
; enter: c = requested drive
; exit:  hl == 0 if drive not allowed
;        else continue to bios seldsk
;
rsxseldsk:
        ld      hl,(vector)     ; shift ok-drives vector left
        ld      a,16
        sub     c
rsxs1:  add     hl,hl
        dec     a
        jr      nz,rsxs1
        ld      hl,0000         ; prime error return
        ret     nc              ; NC if bit wasn't set
jseldsk:jp      $-$             ; jmp to bios seldsk routine
;
;
; Restore this rsx's particular patches.
;
custom_remove:
        ld      hl,(jseldsk+1)  ; restore bseldsk address
        ld      (bios+1ch),hl   ; to bios jmp vector
        ret

;
; *---------------  Standard RSX Code  ----------------------------*
;/                                                                  \
;
; The warm-boot intercept.
;
rsxwb:                          .new
        call    fix0001         ; ensure correct page 0
        ld      hl,(bios+4)     ; does bios wb addr
        ld      de,rsx+3        ; point at us?
        or      a
        sbc     hl,de
        jr      nz,rsxwb1       ; no, we're not the bottom rsx
        ld      hl,(rsxprot)    ; we are, set our protect address
        ld      (0006),hl
rsxwb1: ld      bc,(0004h)      ; get c = logged du for ccp
        jp      rsxnext         ; in case we're top rsx
;
;
; The removal routine.
;
rsxremove:
        call    custom_remove   ; do extra restoration for this rsx
;
        ld      hl,(nextwb)     ; get saved original warmboot addr
        ld      (bios+4),hl     ; and restore it to bios jmp vector
;
; When the caller terminates to a warmboot,
; the next module (or bios, if none), will correct 0006.
;
; Set CY flag to inform the removal tool that this routine
; has indeed taken action. (Some RSX's are not self-removing).
;
```

with this function number, it will set a new vector of valid drives (if DE is non-zero), or report the current vector (when DE is zero). For example, to make drives A:, B: and D: valid, call the BDOS with C = 241, DE = 11 = 1011b.

The RSX must have a module name of the form "RSX...", so that JetLDR can identify it. Following that, the standard header begins the code segment at label "rsx:".

Next come the unique data and code for this RSX, beginning with its null-terminated ASCII name and the vector of valid drives. The static value of the vector is assembled with all drives enabled.

The action begins at "rsxstart". Every BDOS call is intercepted here and tested for this RSX's function number. Most of the time, it will be some other function, and so control jumps to "next"—and then on to either an RSX immediately above this one, or the BDOS. However, when the function number is for this RSX, the routine proceeds to test DE for zero and either load the valid-drives vector into HL, or set the vector to the value in DE. In either case, the RSX returns to the caller.

That's all there is to the added BDOS function. But to make it work, the RSX must also intercept the BIOS select-disk routine, at "rsxseldsk". The code checks the requested drive number against the valid-drives vector, using a simple loop that does a 16-bit left shift into the carry flag. If the drive is not active, it returns the BIOS select error (HL = 0). Otherwise, it continues to the BIOS select-disk routine.

Housekeeping Code

These two routines—"rsxstart" and "rsxseldsk"—do all of the work; the rest is necessary housekeeping. The initialization code is needed only to verify conditions and set up the RSX, so it is assembled, beginning at label "init" in the __INIT__ named-common address space. JetLDR will relocate this code into a working buffer in low memory and execute it there; it takes up no space in the resident system extension. JetLDR relocates the code segment of the RSX, allocating space for it immediately below the command processor or the lowest RSX already in memory.

Initialization involves verifying that the RSX can be correctly loaded, linking the rsx into both the warm-boot and BDOS call chains, and installing any additional BIOS intercepts needed for this particular RSX.

The "initlp" code checks each RSX in memory, beginning with the lowest, to see if an RSX with the same name is already in memory. If so, it (indirectly) calls the routine "custom__twin". This routine

can accept, or reject a duplicate RSX; for OKDRIVES a duplicate RSX would be an error.

Provided no duplicate RSX has caused termination, the search eventually reaches the end of the RSX warmboot chain. If there is no RSX currently in memory, then the "rsxnext" address is set to the ccp entry. However, if one or more RSXs are already resident, the address is set to the warmboot address in the header of the currently lowest RSX.

Next, the RSX's warmboot routine is linked into the BIOS's warmboot chain, and the RSX's BDOS entry is linked into the chain that begins on page-0 with the jump instruction at 0005.

The final initialization step is to call "custom_init". For this particular RSX, that code links the select-disk intercept routine into the BIOS jump table.

Now, look back at the "rsxremove" routine. It is in the code segment, because the remove function must be resident within the RSX. It first calls "custom_remove", to take care of unlinking the BIOS select-disk intercept. Then it unlinks the warmboot intercept and exits with the carry flag set to signal successful removal of the RSX.

There is one more step to unlinking the RSX. When the next warm boot occurs, it will be processed by the RSX immediately above this one, or, if there is none, by the BIOS. In either case, that routine will set a new "protect" address on page-zero at 0006. Of course, this RSX must have an essentially similar routine; it is at "rsxwb". It first calls "fix0001", a precaution that ensures the correctness of the pointer at 0001 to the actual BIOS jump vector.

Next, it determines whether this RSX is, in fact, the lowest RSX in memory; it does this by checking the BIOS jump's address against the RSX address. Only if it is indeed the lowest does it set the protect address on page-zero. Finally, it jumps to the next higher RSX, taking care to first load the current drive/user byte into C, in case the next "rsx" is in fact the command processor.

The details of managing the two linked lists—one pointing upward to higher RSXs, the other pointing initially downward from the BIOS and then to successively higher RSX's—*are* tedious, but necessary. But now, with JetLDR, most of the work can be avoided, and only the particular, custom elements of an RSX need to be specially coded.

```
fix0001:ld      hl,(rsxwba)     ; restore (0001) in case an errant
        ld      (0001h),hl      ; application has tampered with it
        scf                     ; set CY to signal success
        ret
;
;
; Before loading an RSX, JetLDR will first check for protected memory.
; If it detects that memory is protected by a non-RSX header (e.g. a debugger)
; it will cancel the load.  Otherwise, JetLDR will call any
; code in the _INIT_ named common, after the rsx module has been
; loaded and relocated.  This code will be located in non-protected
; memory, and takes no space in the RSX.
;
; Return parameter: A = 0 indicates a good installation
;                   A = ABORT = 0FFh = not installed
;
common  /_INIT_/
;
; Install the rsx.  This code is standard for all rsx's,
; except for:
;       custom_init
;       custom_twin
;
init:   ld      hl,(0006)       ; hl = possible rsx, or bdos
        ld      c,0             ; initialize count of rsx's
;
initlp: push    hl              ; stack (possible) rsx base address
        ld      de,09           ; if candidate is an rsx
        add     hl,de           ; ..the wbaddr will be here
        ld      e,(hl)          ; get address
        inc     hl
        ld      d,(hl)
        ld      hl,(0001)       ; and compare
        or      a
        sbc     hl,de
        pop     hl
        jr      nz,inittop      ; warmboot addr not there, stop looking
;
; we have an rsx in memory, is it our twin?
;
        inc     c               ; count an rsx found
        push    hl
        call    ckname
        pop     hl
        jr      z,twin
;
        ld      de,0Fh+1        ; that rsx was't a twin, check for more
        add     hl,de           ; get addr of next rsx's wboot jmp
        ld      a,(hl)
        inc     hl
        ld      h,(hl)
        ld      l,a
        dec     hl              ; back up to head of that next rsx
        dec     hl
        dec     hl
        jr      initlp          ; now check that rsx
;
; we're at the top of the (possibly empty) rsx chain
;
inittop:
        inc     c               ; any rsx's found?
        dec     c
        ld      hl,ccp+3        ; prepare to use ccp entry address
        jr      z,setnext       ; ..no
;
        ld      hl,(0006)       ; yes, use bottom rsx's address
;
setnext:
        ld      (rsxnext+1),hl  ; save the next addr
                                ; in the rsx chain to bdos/ccp
;
; install the rsx into the running system
;
        ld      hl,(bios+4)     ; save the bios's wb addr
        ld      (nextwb),hl     ; in the header
        ld      hl,rsx+3        ; point the bios wb jump
        ld      (bios+4),hl     ; at the rsx wb vector
```

```
        ld      hl,bios+3       ; store wb addr
        ld      (rsx+09),hl     ; in rsx header word
        ld      hl,(0006)       ; get addr of next rsx or bdos
        ld      (next+1),hl     ; and install it
        ld      hl,rsx          ; finally, protect the rsx
        ld      (0006),hl
;
        call    custom_init     ; take care of extras
        ret
;
ckname: ld      de,0dh          ; offset to candidate rsx name pointer
        add     hl,de
        ld      a,(hl)          ; get address
        inc     hl
        ld      h,(hl)
        ld      l,a
        ld      de,rsxname      ; compare to our name
ckname1:ld      a,(de)
        cp      (hl)
        ret     nz
        inc     (hl)            ; candidate must be nul-terminated
        dec     (hl)
        jr      nz,ckname2
        or      a               ; ..at our same byte
        ret
ckname2:inc     hl
        inc     de
        jr      ckname1
; Handle the case of a previously-loaded copy of this RSX.
;
twin:   call    custom_twin
        ret
;\                                                              /
; *--------------------------------------------------------------*
;
; Custom initialization code goes here.
;
;
; Do the particular patches for this RSX.
; Note: this code is in the _INIT_ segment.

custom_init:
        ld      hl,(bios+1ch)   ; get bseldsk address
        ld      (jseldsk+1),hl  ; install it in rsx
;
        ld      hl,rsxseldsk    ; divert bios jump
        ld      (bios+1ch),hl   ; to the rsx
        ret
;
; This particular rsx should not be duplicated.
; A different rsx might wish to query the user here,
; print a warning, or whatever.
;
custom_twin:
        ld      a,ABORT
        ret
; Include identification info in the REL image.
; JetLDR will display the bytes up to the first NUL byte
; when the RSX is loaded.
;
;
common  /_ID_/
;
        db      'OKDRIVES: RSX prevents bios logins'
        db      13,10
        db      'Use BDOS function 241 (0F1h) to set de = drive vector',0
;
; Include whatever other named-commons are needed for this RSX.
; JetLDR will resolve these labels for us.
;
common  /_BIOS_/
bios    equ     $

common  /_CCP_/
ccp     equ     $
        end     ;okdrives.asm
```

Extending System Extensions

As the wag said, "I like standards, because there are so many to choose from!" The Plu*Perfect Systems RSX header is just one way to add resident extensions to CP/M 2.2. But it is well tested, provides for BIOS as well as BDOS system extensions, and takes care to be compatible with other RSXs.

I have used these RSX structures in BackGrounder ii—as well as its spooler, printer redefinition module, and secure memory allocator—in DateStamper, and in DosDisk, achieving a high degree of compatibility between them. More recently, Carson Wilson, Joe Wright and others have adopted the Plu*Perfect Systems header structure to add such extensions as quad-density disk drivers to a BIOS and resident conditional-execution processing (IF.COM) to ZCPR34.

This experience shows that a uniform RSX header can, indeed, allow diverse programs and applications to work together. Several older programs, including versions of BYE for remote operation of a CP/M system, and ZEX for in-memory submit processing could be revised to use this interface, so that other RSXs could be run while those extensions are active.

In a CP/M 2.2 system that includes additional memory, it is possible to make system extensions "resident" without subtracting from scarce memory for applications. Malcom Kemp has pioneered this approach, called Banked System Extensions, by defining a similar BSX header and providing loading/removal service in the XBIOS system for the SB180 and SB180/FX computers. I developed both a banked-memory DateStamper and a DosDisk for this system. In fact, I am completing this column on an XBIOS system with banked DateStamping and a DOS disk in drive C:, with no loss of TPA!

The next time you wish your computer had some missing capability, consider whether it might be added as an RSX. Those features may already be available in products such as DosDisk, BackGrounder ii, or Z3PLUS. If not, you may be able to code the routines yourself and bring your system to new levels of performance and versatility. ■

Data File Conversion

Converting Macintosh Files with Turbo Pascal

by Tim McDonough

Art Carlson's recent article on converting foreign data files *(Data File Conversion, Writing a Filter to Convert Foreign Formats.* TCJ Issue #33) got me to thinking. How many times does one of us solve some application problem and then take the solution for granted without ever bothering to share it with the other users around? Fairly often I would guess.

A few months ago, my wife and I contracted with a local publisher to input the data for thousands of mailing labels that were to be used in a direct mail campaign. We have both Macintosh and CP/M machines available. The publisher uses a Commodore C-128 to run his local office. It was unimportant to the customer what type of machine we used to enter the data as long as he ended up with a file that had a fixed number of fields per record, using only a carriage return (ASCII 13) as a field separator and a record delimiter.

The program that we normally use for day to day operations on the Macintosh is *Microsoft Works.* This program combines word processing, spreadsheet, database managment and telecommunications into a single integrated package. What Works doesn't do is give the user many options when it comes to exporting database files for use in other programs or other machines. If you instruct Works to export a file, it creates a plain ASCII text file that has a tab character at the end of each field and a carriage return at the end of each record.

The problem was simple. What we needed was a program or filter that would search the exported file for tab characters and replace them with carriage returns. No problem, except that if you've ever seen or used a Macintosh before, you will recall that there is no command line interface. Instead there is a sort of electronic desktop with pull down menus, icons, mice, etc. that make the machine quite unlike a CP/M or MS-DOS computer. This is not a tutorial on Macintosh programming so I won't delve into a lot of detail but let's just say that in order for a normal Macintosh application to be so user friendly the programmer has to go to great lengths to craft a well behaved program with all of the features Macintosh users have come to expect.

The key things in the preceding paragraph are normal and well behaved. This filter program we need to write is very limited in its use. Like Art Carlson's PRDB.C, it would only be used a few times and then archived.

One of the things that Apple never mentions too often is that the sophistication of the Macintosh lies in the hundreds of carefully crafted routines that are a part of the Mac ROMs and that the computer itself is a fairly traditional, 68000 based design. It is possible, although generally not desireable from the user's point of view, to write a plain vanilla computer program on the Macintosh. In fact most of the Macintosh software development systems such as Borland's Turbo Pascal make this type of program just as easy to write on a Mac as it is on a CP/M based computer.

The solution to our conversion problem is just such a program. Listing 1 is the Turbo Pascal source code for TABCon.pas, the program that converts tabs to carriage returns. When the program is run, it prompts the user for two file names and then procedes to convert the input file, displaying a running count of the number of records processed as it goes. When the program ends, the user id returned to the Apple Finder program, which is what nearly all normal Macintosh applications do when the user exits.

The Pascal Program

The first block of most Pascal programs is the *const* block where constant declarations are made. Constants are used to improve the readability of a program and make software maintenance easier when necessary. Two constants are decalared, TAB and CR. The symbol TAB is given the ASCII value 9 and CR is given the ASCII value 13.

The next block of the program is the *var* block which contains the programs global variable declarations. RawText and AsciiText are decalred to be of type *text. Text* is the Pascal type that refers to a file of ASCII characters. InFile and OutFile are decalred to be type *string.* These variables will be used to hold the names of the input and output files involved in the conversion. The variable C is declared to be of type *char.* This variable is used to hold a single character as it is processed by the program. Next to be declared is an *integer* variable count. Count is used to hold the value of the running counter that is displayed to show the conversions process along the way. Finally, the variable valid is decalred to be of type *boolean.* A boolean variable can be only one of two states, true or false, as its name implies. Valid is used as a flag to indicate the result of some crude error checking on the file names supplied by the user.

The remainder of TABCon.pas consists of two Pascal procedures and a short main program. The first procedure, *initialize,* clears the Mac's screen, displays some static text and messages and prompts the user for the names of the input and output files. Before trying to open either file, the input is checked and if the user has pressed the RETURN key in response to either prompt, the flag *valid* is left set to false and neither file is opened. The second procedure, *cleanup,* checks to see if the *valid* flag is true and if it is the files are closed and a message printed to the user. These procedures perform no action by themselves and must be called from the main program in order to have any effect.

LISTING 1

```
program TABCon;
{
  Program:      TABCon.pas

  Language:     Turbo Pascal

  Environment:  Macintosh

  Author:       Tim McDonough
                Cottage Resources
                Suite 3-672
                1405 Stevenson Drive
                Springfield, IL 62703

  Date:         December 4, 1987
```

```
  TABCon was written to remove the TAB characters used by
  Microsoft Works to seperate fields in exported databases. It
  replaces every occurence of a TAB charcater with a carriage
  return.

  T. McDonough December 4,1987

}

const
  TAB = 9;
  CR = 13;

var
  RawText, AsciiText : text;
  InFile, OutFile : string;
  C : char;
  count : integer;
  valid : boolean;

procedure initialize;
begin
  count := 0;
  ClearScreen;
  Writeln( 'ASCII Conversion Program -- <TAB> -> <CR>' );
  GotoXY( 1, 5 );
  write( 'Input file name:' );
  GotoXY( 1, 7 );
  write( 'Output file name:' );
  GotoXY( 1, 9 );
  write( 'Records processed:' );
  GotoXY( 21, 5 );
  readln( InFile );
  valid := FALSE;
  GotoXY( 21, 7 );
  readln( OutFile );
  if ( InFile <> '' ) and ( OutFile <> '' ) then
    begin
      reset( RawText, InFile );
      rewrite( AsciiText, OutFile );
      valid := TRUE;
    end;
end; (* of procedure *)

procedure cleanup;

begin
  if valid = TRUE then
    begin
      close( RawText );
      close( AsciiText );
    end;
  GotoXY( 1,11 );
  Write( 'Conversion complete. ' )
end; (* of procedure *)

begin
  initialize;
  while not eof( RawText ) do
  begin
    read( RawText, C );
    if C = chr( CR ) then
    begin
      count := count + 1;
      GotoXY( 21,9 );
      Write( count:10 );
    end;
    if C = chr( TAB ) then
      C := chr( CR );
    write( AsciiText, C );
  end;
  cleanup;
end.
```

The last few lines comprise the main body of the Pascal program. First the initialize procedure is called which opens the files as described above. Next a loop is executed as long as the the intrinsic function, *eof()*, does not return the value true. The loop performs the following actions: A character is read from RawText; if the character is a carriage return then the counter is incremented and displayed; if the character is a tab then its value is converted to that of a carriage return; finally, the character either as originally read or after it is converted to a carriage return is written to the output file. This process happens again and again until the value of eof(RawText) becomes true signalling the program that the end of the input file has been reached. Finally the procedure cleanup is called to close the files as described above.

The very last line of the program is the word *end* followed by a period. This signifies to the Pascal compiler that this is the end of the main body of the program and the source file.

Loose Ends

The TABCon.pas program doesn't entirely solve the conversion process. Converted or not, stuffing 3.5 inch Mac disks into a 5.25 inch Commodore C-128 disk drive doesn't work very well. Aside from the physical incompatibility, the Mac's drives hold about 800K of data, much more than a standard Commodore floppy. The solutions to both these problems involved no programming. The mailing list databases were broken up into chunks of about 50K bytes each. This size was chosen by the publisher to make them easy for his mailing list management software to manipulate. The physical incompatibility was solved by Minnie, a local multi-user bulletin board system. As we finished our data entry and format conversion of the files, they were uploaded to the publisher's private directory on the BBS. He then downloaded them at his convenience. If you use such a process for your own work, check with the sysop first. We are fortunate in our area to have a local system with several hundred megabytes of online user storage and multiple incoming phone lines. The transfer could have been done directly to the publisher's computer but it was more convenient to use the BBS. Transfering hundreds of thousands of bytes to small PC-based single user system might not win you any friends. ■

Back Issues Available:

Issue Number 1:
- RS-232 Interface Part One
- 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 Two
- 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 the 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 6:
- Build High Resolution S-100 Graphics Board: Part 1
- System Integration, Part 1: Selecting System Components
- Optronics, Part 3: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

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 14:
- Hardware Tricks
- Controlling the Hayes Micromodem II from Assembly Language, Part 1
- S-100 8 to 16 Bit RAM Conversion
- Time-Frequency Domain Analysis
- BASE: Part Two
- Interfacing Tips and Troubles: Interfacing the Sinclair Computers, Part 2

Issue Number 15:
- Interfacing the 6522 to the Apple II
- Interfacing Tips & Troubles: Building a Poor-Man's Logic Analyzer
- Controlling the Hayes Micromodem II From Assembly Language, Part 2
- The State of the Industry
- Lowering Power Consumption in 8" Floppy Disk Drives
- BASE: Part Three

Issue Number 16:
- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a "Jellybean" Logic-to-RS232 Converter

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
- The Computer Corner

Issue Number 19:
- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC
- The Computer Corner

Issue Number 20:
- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K
- The Computer Corner

Issue Number 21:
- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC
- The Computer Corner

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
- The Computer Corner

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 Index
- The AMPRO Little Board Column

Issue Number 24:
- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly 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 Realtime 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 Little Board
- 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 and 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 for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats
- The Computer Corner

Issue Number 28:
- Starting Your Own BBS: What it takes to run a BBS.
- Build an A/D Converter for the Ampro L.B.: A low cost one chip A/D converter.
- The Hitachi HD64180: Part 2, Setting the wait states & RAM refresh, using the PRT, and DMA.
- Using SCSI for Real Time Control: Separating the memory & I/O buses.

- An Open Letter to STD-Bus Manufacturers: Getting an industrial control job done.
- Programming Style: User interfacing and interaction.
- Patching Turbo Pascal: Using disassembled Z80 source code to modify TP.
- Choosing a Language for Machine Control: The advantages of a compiled RPN Forth like language.

Issue Number 29:
- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a nes OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner
- The Computer Corner.

Issue Number 30:
- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR# Corner
- The CP/M Corner
- The Computer Corner

Issue Number 31:
- Using SCSI for Generalized I/O: SCSI can be used for more than just hard drives.
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.
- The Computer Corner

Issue Number 32:
- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. 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: Part 2, 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, and how to write self relocating Z80 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 and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

TCJ ORDER FORM

Subscriptions		U.S.	Canada	Surface Foreign	Total
6 issues per year ☐ New ☐ Renewal	1 year	$16.00	$22.00	$24.00	
	2 years	$28.00	$42.00		
Back Issues		$3.50 ea.	$3.50 ea.	$4.75 ea.	
Six or more #'s		$3.00 ea.	$3.00 ea	$4.25 ea.	
				Total Enclosed	

All funds must be in U.S. dollars on a U.S. bank.

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____ State_____ ZIP _____

The Computer Journal

190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

MOVING?

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

If you move and don't notify us, TCJ is not responsible for copies you miss. Please allow six weeks notice. Thanks.

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these registered trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used marks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosch, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder ii, DosDisk; Plu*Perfect Systems; Clipper, Nantucket; Nantucket, Inc. dBase, dBase II, dBase III, dBase III Plus; Ashton-Tate, Inc. MBASIC, MS-DOS; Microsoft. WordStar; MicroPro International Corp. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C; Borland International. HD64180; Hitachi America, Ltd. SBI80 Micromint, Inc.

Where these, and other, terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.

FOR SALE

Apple II Plus, 64K RAM, two Apple Disk II drives, Apple drive controller card, Epson parallel printer card, Zenith green monitor, Original Apple manuals, Word processor, Database system, Assembler, books. All software original disks, not pirated... **$525**

Vista A800 8 inch double density disk controller for the Apple II+. Complete in box with manual and patch disks for DOS 3.3. I used it with a Shugart 850 DSDD drive..... **$120**

John Bell Peripherals for the Apple II+ and IIe. 6522 Parallel interface #79-295A, A-D Converter #81-132A, Eprom Programmer #80-244A with Textool socket. All boards factory assembled. Total for all three boards..... **$100**

Shugart 860 8 inch half height double sided drive. In box, unused..... **$250**

Zenith Z-19 Terminal. Needs work (video board?)..... **$95**

Davidge DSB-4/6 Z80 single board computer. Nice litte board, runs both 5.25 & 8 inch drives. With system/utilities disk and manual..... **$85**

AMPRO Little Board Bookshelf unit. Z80 CPU, half height 5.25 double sided drive, 10 Mbyte hard drive, SCSI port, ZCPR3. Up and running, works perfect, but I have four of them. Good BBS base unit. With system disks and manual..... **$700**

Morrow Micro Decision 2.2 Motherboard in Morrow case with power supply—no drives. Manuals and disk included..... **$150**

Tektronix 4023 graphics terminal..... **$95**

All prices plus shipping.

The Computer Journal
190 Sullivan
Columbia Falls, MT 59912
Phone (406) 257-9119

(Continued from page 44)

MSDOS programs as well. The needed additions for MSDOS are more memory and the compatible video controller card.

Although this sounds good, my main interest was bringing up CP/M 68K using all the same cards except the CPU. I have tried a few times so far, but have not found all the problems yet. GodBout had configured the CP/M 68K for their Disk 1A and I am using a disk 1. The main difference is how they use ROM space at boot time. The disk I has only 256 bytes available, while the 1A has more. This means I have to put my boot ROM elsewhere and have the disk 1 jump to it.

This is where my playing has left me at present, burning new ROMs and trying it all out. Since moving and rebuilding my computer room are taking priority it will be some time before I can report on this project.

A Last Quick Shot

Another project has appeared on the horizon, writing a book. As with all new books out, this one has to do with the PC systems. This forced me to buy another PC clone, one more closely related than my favorite Heath/Morrow 171/Pivot. I tried a desk top publishing system on the 171 and got no where using the keyboard (might work with a mouse—will try that later). I have also been using an extension box with the 171 and found it unable to reset to a hard disk on crashes (must kill all power to get HD working properly again). The last problem with the 171 is the CGA screen, which is just unacceptable for desk top publishing.

The solution is a newer PC clone with 15 Meg of hard disk. I got all of that last weekend at a swap for $475. It includes Hercules graphics, which I like better than CGA and EGA, as I am not that crazy about color. I did have some problems which I am sure is typical for many users. I backed up the hard disk on the 171 using utilities I had loaded on it some time ago. The new PC had version 3.21 of DOS, the same version that was suppose to be on the 171. Well as you can likely guess, when I went to restore my old data to the new hard disk it didn't work.

I would load the disks into the system and it would prompt me to add disks and blink like it was working. I thought something was wrong because it took only 5 minutes to load all the 18 disks. Checking the directory showed that nothing happened. I watched the whole routine again and all that was happening is the system read the disk and checked the directories and for some reason decided that none were usable and so asked for the next disk in order. When I used the same restore from the disk that I got the backup from the program worked correctly. I also noted that it displayed a new message indicating that I was restoring data to a fixed disk.

In trying to figure out whether or not I was doing something wrong I checked the Heath/Zenith manuals and their versions of backup and restore. I like theirs better as it contains a help screen as well as telling you which version of the program you are using. The official backup and restore give you no idea which version you are using and as such problem like I describe are all to easy to occur. If most readers are like myself, you too may have 6 to 10 versions of DOS floating around, all with possibly different versions of utilities.

The answer to these problems is proper storing and cataloging of versions by the user. I need to go through and make sure all systems contain exactly the same DOS and utilities. MicroSoft on the other hand could make life easier by adding prompts to their programs that tell what version you have just loaded. It also would have helped if some form of error message could be displayed saying that I was trying to use an incompatible version of restore. As I think about all this, it reminds me of how people never learn. I can remember saying things like this eight years ago with some of the CP/M programs just then out. Oh well, two steps forward and three back...

That also reminds me of the times past, when finding time wasn't so hard. Right now I am so busy and have so many plans, like contacting Joe Bartel at the SOG next week, that it gets hard for me to find the time to write. Hopefully the SOG will give me plenty to write about. I also hope that my systems will be back together as they are now in pieces in temporary storage. ■

THE COMPUTER CORNER

by Bill Kibler

BUSY, BUSY, BUSY. This article catches me on the run and then some. We are in the process of moving to a fixer up house with some land and have been finding everything needing fixing. Toss in a pending trip to the SOG in Oregon and that leaves little time for computing.

I do have some words of wisdom and some experiences to relate, and they start out with Forth. Since I bought the Novix Forth engine, I have become interested in other such machines. A friend of mine gave me his Rockwell Forth evaluation system. This machine uses the 65F11 or 65F12. These devices are 6502s, with internal ROM, timers and I/O ports. The ROM contains a small Forth kernel and can be brought up to run your application without any other Forth code.

The development system does have an extra 2716 EPROM that contains utilities for accessing a disk drive and reading and writing screens. The development ROM is still in need of more work to provide all the features of a normal Forth, but what you do get is enough to get the system into working shape quickly.

My interests were in building a demo module to show how Forth could be used to control hardware. The idea is to have it run a mock elevator system. A stepper motor would run it up and down, while switches control its stopping at the proper floor. So far I have found a few bugs in the system, which is why I got the unit in the first place. The manual shows a timer routine that uses the interrupt ability. The screens of code will not work as listed. I tried them and they would lock the system up every time. The solution was minor but makes me worry about the overall design. Rockwell added some new instructions to help in changing bits in memory locations. Using these instructions will lock the unit up as will using a push or pop inside of the interrupt routine. By not using the extra instructions and saving your registers to allocated memory (in place of push and pop) the program works most of the time.

This problem has stopped my project currently, until I get a chance to contact a company in Canada. The Forth dimensions has a ad from the Canadian company who has just released a new product using the chips. I hope they have found all the problems and have fixes for them. I will let you know more about the company and the problems after talking with them.

RTX2000

At the last local Forth meeting, a sales rep for Harris devices gave a talk on the RTX2000. The RTX is the Novix core with several other ASIC devices added on. Harris has bought the rights to use the Novix Forth core in their ASIC library of devices. The rep said that several big companies have already used the Novix core for their special applications. The RTX is the first product from Harris in which they have a general purpose approach. They promise to produce several more versions with various special features and uses in mind.

The RTX2000 is their Real Time eXpress device. The market is real time applications such as graphics, CNC machines, and whatever you can dream up. The device consists of the Novix core, static RAM for the stacks, a stack controller, a memory management unit, interrupt controller, timers, and a ASIC bus interface. The Novix needed external stacks, so the RTX's internal RAMs save 48 pins. The RTX is housed in a 84 pin array compared to the Novix's 121 pins. There are also 18 address lines for 1 Meg of memory (all 16 bits wide).

Harris has cleaned up the entire Novix and solved a few problems from the past. Their added devices make the unit perfect for many applications that would be hard to wire together using the Novix. I am already hot to put it a few places, even at $200 dollars a pop presently. Their are some problems, like the 10 Megahertz speed. At that speed you will need 50 NS or better static RAMs for memory. They have added a wait circuit which will help with slow RAMS and the memory cost, but you will be hard pressed to get a real fast system for less than $2000.

I didn't really cover speed, but fast it is. The 10 megahertz should produce about 15 MIPS, that is 15 million instructions per second. One of the members commented that a company back east supposedly emulated the 80286 on a Novix (6.5 MIPS) and was able to run programs at twice the speed of the AT type systems. Now I don't have any facts to back that up, but my own Novix is so fast it can do everything with time to spare. It is currently housed in a clear plexiglas® box to show off the very few parts involved. In software it does all the reading and writing of the disk drive, reads the PC compatible keyboard, and supplies all signals to a TTL PC compatible monitor. My system all fits into the 8 by 9 by 9 inch box, including the monitor, but not the keyboard.

Harris has two ways you can play with their RTX, a PC development board and a stand alone starter board. The PC unit goes for about $4000, less if you buy more, and comes complete with software (lots, include a C compiler) and manuals. The software allows using the DOS as I/O. The stand alone has a monitor and serial interface system for downloading screens from a host PC. The stand alone cost is about $1500 and we figure that is cheaper than the cost of the parts separately (remember it uses 50NS static RAMS!).

I will keep you informed about the device as more information becomes available. If I can find a buyer for my Novix in a plexiglas box system, I will buy a RTX and make a system around it. Actually I think these are Christmas dreams in the heat of the summer.

S100 AGAIN?

Yup, I got some newer S100 cards and have been playing a little with them. I recently bought a GodBout S100 system for $100. Now that is 2 eight inch drives (double sided) and a complete CPU 8085/8088 system. It is now up and running (had a few bad devices and jumpers) and works rather nice. You can jump between CP/M 85 and CP/M 86 programs just by typing their name. If I had the money or time I could even get their new concurrent CP/M and run

(Continued on page 43)