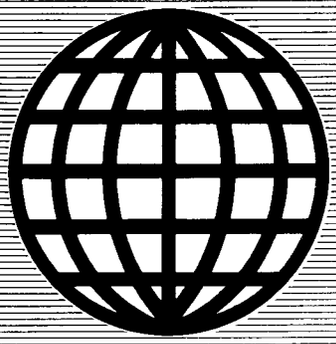


The COMPUTER JOURNAL®

Programming - User Support

Applications

Issue Number 41 November / December 1989 $3.00

Forth Column
ADT's, Forth and Object Oriented Concepts

Improving the Ampro LB
Discard the 88Mb Hard Drive Limit

Data Structures in Forth

Advanced CP/M

The Z-System Corner

Programming Input/Output With C

LINKPRL
Making RSXes Easy

Real Computing
The NS320XX

SCOPY
Selective File Duplication

ISSN # 0748-9331

The Computer Journal

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Bridger Mitchell
Clem Pepper
Richard Rodman
Jay Sage
Dave Wenstein

The Computer Journal is published six times a year by Publishing Consultants, 190 Sullivan Crossroad, Coulmbia Falls, MT 59912
(406) 257-9119

Entire contents copyright © 1989 by Publishing Consultants.

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 U.S. dollars on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912, phone (406) 257-9119.

The COMPUTER JOURNAL

Issue Number 41 November / December 1989

Editorial 2

Forth Column 3
Implementing abstract data types, and object oriented concepts. By Dave Weinstein.

Improving the Ampro LB 6
Overcoming the 88Mb hard drive limit. By Terry Pinto.

Data Structures in Forth 9
How to add data structures. By Joe and Marla Bartel.

Advanced CP/M 12
CP/M is still the Hacker's Haven. The Z-System Command Scheduler offers new capabilities. By Bridger Mitchell

The Z-System Corner 16
The Extended Multiple Command Line and more on aliases. By Jay Sage.

Programming Input/Output With C 20
Part 2 continues with the disk and printer functions. By Clem Pepper.

LINKPRL 27
Making RSXes Easy. By Harold F. Bower.

Real Computing 30
By Richard Rodman.

SCOPY 32
Copying a series of unrelated file names is a chore, but the job is simplified with SCOPY. By Dr. Edwin Thall.

Computer Corner 40
By Bill Kibler.

Editor's Page

Forth

We initiated our Forth section in the last issue with two articles from Dave Weinstein. It continues in this issue with another article from Dave, who will be a regular contributor, plus and article from Joe and Marla Bartel. We already have material lined up for issue #42, and intend to continue the Forth section in every issue.

There has been a lot of activity in the Forth community, and not everyone agrees with some of the changes. One of the changes which I appreciate is the use of regular ASCII text files for programs instead of screens and blocks. Some people object to the "Maxi-Forth" concept (such as F-PC) and are more comfortable with the traditional "Mini-Forth." Most of my programming (the little that I have time for) involves filtering ASCII text files or accessing dBASE DBF files, so I prefer the implementation which uses and accesses the regular system file structure. I am also more comfortable with a full screen ASCII text editor for the program where I can look at the overall program flow instead of all the separate screens and blocks. Others who program embedded real time systems may prefer to continue to deal with the traditional Forth structure. There is room for both concepts.

One of the things which turned me off about Forth in the past was the fact that the literature concentrated on Forth itself, and there was little information on how to actually do something with it. As Dreas Nielsen (whose Forth article on Dynamic Memory Allocation is scheduled for issue #42) stated, "... an exceptional proportion of the Forth articles appearing in the literature are about modifications to Forth itself—new vocabulary structures, word headers for accumulating profiling statistics, etc. This kind of information is rarely of any interest, and more importantly, not of any use to the casual or would-be user of Forth. I hope that your articles will emphasize applications instead."

We intend to concentrate on practical examples demonstrating what can be done with Forth, and how to do it. This will, of course, involve Forth modifications, but we also want to include a lot of "how-to" application examples to interest new users in learning to use Forth. I have a hard time in using Forth because I don't know how to tackle my applications. I usually use C for the simple text filtering programs I use to illustrate articles (see the DTP and LaserJet articles in recent issues) and one of the challenges for our Forth authors is to get me started in converting these examples to Forth.

Dreas also suggested that we sponsor a competition to produce a set of dBASE file access functions in Forth using Forth words similar in name to the dBASE features. I think that is a great idea. Creating these functions will provide a very useful tool for database applications, while serving as a wonderful learning tool, and we can provide the library in both hard copy and low cost public domain disks. Anyone interested in getting involved in this?

Let us hear about your Forth comments and questions—what do you want to see in the Forth section?

Hot Drives

Some time ago I described ST-225 drive problems which were caused by excessive heat. Since then I have run across numerous mentions of others with similar problems.

Recently the ST-251 drive in my '286 system has started making a high-pitched noise after being on for a while. The drive cage had closed sides which prevented air circulation and the hard drive became quite warm. The only forced ventilation in the system is the exhaust fan in the power supply, and it does very little towards cooling the drives.

The real problem is that the drive cage which holds three drives makes absolutely no provisions for cooling. That's OK for floppies which only run intermittently and generate little heat, but 5¼ hard drives generate a lot of heat and need a flow of cooling air. I drilled some air holes (after removing the drives from the cage, and the cage from the main box) in the side of the cage and mounted a small fan on the cage. It helps, but there is so little space between the drives that it is difficult to force enough air between the drives.

I'm still looking for a better solution without replacing the entire box. I'm considering moving one or both of the floppies to an external box so that there is enough air space around the hard drive. I hate the thought of another box with cables—but I need to keep the hard drive cool.

If you have hard drive problems, consider the effect of the lack of cooling. I'd like to hear about your experiences. I've used this drive for a year and I feel that it is time to back up all the files and do a complete low-level reformat. This would also be a good opportunity to clean up the directories and get rid of a lot of files which I don't use very often.

Users Must Program

To many people it is obvious that software should be designed and programmed by a 'professional' programmer, and that it is the user's responsibility to modify their application to agree with the software—and to live with any shortcomings in the programs.

There are a few mavericks who do not agree. A scientist recently stated, "I've never seen a useful program which was not written by a scientist or an engineer." He went on to say that it was easier to teach a scientist to program than it was to teach a programmer what the working scientist required.

I have always felt that it is necessary to be able to program if you want something done your way. I don't mean that we should all be able to write our own spreadsheet, wordprocessor, or Desk Top Publishing program. I wouldn't even attempt something like Microsoft Word, PageMaker, or one of the many drawing programs--they would take too much effort. But we can write our own text file filters, printer drivers and runoff programs, and other modest sized programs to meet our specific needs.

We will increase our coverage of algorithms (methods of attack) for program solving. We'll try to include pseudo-code so that it will be meaningful even if you don't use the particular programming language used in the illustrations. We'll spend more time discussing application problems and approaches than on neat coding tricks.

What would you like to see covered? Mail in your suggestions and/or articles. •

Forth Column

ADT's, Forth, and Object Oriented Concepts

by Dave Weinstein

An Abstract Data Type (ADT) is a description of an object. For example, given an abstract description of a stack, I can act upon the stack without knowing anything about the implementation, so long as I use the legal actions as described (see Figure 1). The concept behind this is to separate the implementation from the concept. This allows the programmer to concentrate upon using the data structure, rather than upon how the structure works (in practice it pays to keep the implementation in mind, but not to act on this knowledge).

The modules provided in the last column allow the programmer to build these data structures, and protect them from casual use. But there is nothing in the modules which prevent the programmer from using the internals of the data structure outside of the module in which the structure was designed. So, for example, there is nothing other than good programming practice to keep another module from tweaking the internals of a stack. The ability to "break" layers of abstraction has had computer science theorists up in arms for some time, but paternalistic languages (such as Modula-2) which enforce this kind of abstraction also tend to confine Forth programmers, who are used to being able to do just about anything. Since it is possible to write bad code in any language (not only possible, but judging by most of the code extant, also probable), I find enforcing abstraction, especially in a language like Forth which depends upon the skill and knowledge of the programmer to keep the program from leaping off of the nearest cliff, to be in general counterproductive.

Figure 1: Stack ADT Design

Initialize -- (size ^stack --)
: Initialize the referenced stack so that it has the specified size.

Empty? -- (^stack -- f)
: Return a flag as to whether or not the given stack is empty.

Full? -- (^stack -- f)
: Return a flag as to whether or not the given stack is full.

Push -- (value ^stack --)
: Push the given value onto the stack.

Pop -- (^stack -- value)
: Pop the top of the stack and place the result on the parameter stack.

Top -- (^stack -- value)
: Copy the top of the given stack and place the result on the parameter stack.

Clear -- (^stack --)

An Example Using Forth

Figure 2 provides the code to implement a stack ADT in Forth, based on the specifications from Figure 1. Each stack in this implementation will consist of three pointers, (the ^ sign in Forth code should be pronounced "pointer" if it comes at the end of a word, and "pointer to" if it comes at the beginning). The bottom and top pointers are used to figure the boundaries of the stack (this stack grows downwards), while the current pointer points to the next free element.

The word initialize saves the current here pointer, which is going to be the bottom of the stack, and then allocates as much space as was requested (this particular implementation uses stacks with a width fixed at cell...it is possible to change the implementation to use variable width stacks by storing the width in the stack record, and changing the words used to modify current^ [cell- and cell+ are used] with words such as width- and width+ which use the specified width). The address of the last (i.e. top) element of

Figure 2: Code to implement a stack ADT

```
\                       Forth Stack ADT
\
\               written by David Weinstein -- 6/25/89
\
module stacks
        from records import record as record
                     import element as element
                     import end-record as end-record
        end-imports

        record stack
                cell element current^
                cell element top^
                cell element bottom^
        end-record

        ( Operations on the Stack ADT )

        : initialize    ( size ^stack -- )
                here over bottom^ !
                swap allot
                here cell- swap over over
                top^ !
                current^ !
                ;

         : clear        ( ^stack -- )
                dup top^ @ swap current^ ! ;

         : empty?       ( ^stack -- f )
                dup current^ @ swap top^ @ = ;

         : full?        ( ^stack -- f )
                dup current^ @ swap bottom^ @ = ;

         : push         ( element ^stack -- )
                swap over current^ @ !
                0 cell- swap current^ +!
                ;

          : top.of.stack        ( ^stack -- top-of-stack )
                current^ @ cell+ @
                ;

          : pop         ( ^stack -- element )
                current^ dup @ cell+ dup @
                -rot swap !
                ;

end-module
```

the stack is then computed, and the top^ and current^ are set to top leaving the stack empty).

The word clear copies the top^ into the current^, effectively popping and dropping the contents of the stack. The dup's which are floating around serve to save the pointer to the stack record (remember that the indexes in a record add themselves to a base address).

The word's empty? and full? compare current^ with top^ and bottom^ respectively, if they match, then the stack is either empty or full, if they do not, it is somewhere in between.

The last three operations are the most common, push which copies the element into the location pointed to by current^ and then decrements current^, pop which fetches the value pointed to by current^ and then increments current^ to point to the next value, and top.of.stack, which copies the value pointed to by current^ to the parameter stack.

What Does All This Theory Have to do With Forth?

Forth is a weakly typed language with rigidly typed operators. Although variables in Forth are typed according to size, all of the operators are typed according to content. Each additional "type" gets its own addition operators, multiplication operators, and so on. And to make matters worse, operators must be provided for mixed operations. Unfortunately, this sea of operators obscures the code. Although I the programmer have no problem discerning the difference between opening a file and opening a window, Forth as it comes out of the box does. I have to write words like FILE-OPEN, or WINDOW-OPEN, which hide the elegance of the code in verbiage. One option is to go the route of Fortran, C, and Basic, and overload the operators. The burden of understanding what needed to be done and how it should be done would be in the code handling the operator, not the code in which the new type was defined. By adding a "type stack" to Forth this could be done, but at the cost of typing the language (a step many do not want to take).

Object Oriented

This has been one of the biggest buzzwords in the programming community in recent years. And much like Forth, is touted as the ultimate panacea for all programming problems by its aficionados, Object Oriented techniques are now being declared the be-all and end-all of programming techniques by over zealous supporters (so Object Oriented Forth should be unstoppable). Adding to this confusion, and to a large degree capitalizing on it, are vendors who are declaring any language extension which includes any of the object oriented techniques to be an "Object Oriented Language." So what exactly is an object oriented language? With all of the conflicting claims, vendors would have you believe that almost everything is "object oriented" these days. So let's go back to the basics. The core idea of "object oriented" is inheritance, the concept that not only can structures be related to other structures, they can in fact "inherit" the features of their "ancestors" (saving code duplication). Single inheritance systems allow any parent to have any number of children, but any child class can only be directly descended from a single parent (although it will also inherit those classes which its parent inherits). In multiple inheritance systems, child classes can inherit from many classes (allowing more flexibility).

Figure 3: Instances, Objects, and Classes

Class - The definition of an object oriented structure.
Object - An object is an "instance" of a class. If a class is Window, then the DialogueWindow would be an instance of it.
Instance Variable - A special type of variable defined inside of a class. A variable inside of a class definition is common to all objects (i.e. instances of the class), but each object has its own copy of an instance variable, which other instances of the same class cannot access.
Message -A structure which is sends itself to an object, which then decides what to do.
Method - The portion of an object definition which defines how a message is handled.

Figure 4: Object Oriented Extension Syntax

```
Defining a message type:

MESSAGE <name>

(Note that all message names are common)

Defining a class:

CLASS <name>

      SUBCLASS-OF <name>
            ...
      SUBCLASS-OF <name>

      VAR <instance-name>        ( Cell sized instance variable )
      CVAR <instance-name>       ( Character sized instance variable )
      DVAR <instance-name>       ( Double sized instance variable )
   n  INSTANCE <instance-name>   ( n sized instance variable )

        ( any wanted Forth code )

      METHOD: <message-name>
                    ...
         ( Forth code to implement the method )
                    ...
      END-METHOD

  END-CLASS <name>

Allowing code to send a message to itself:

        SELF <message>

Defining an instance of a class:

         <class name> <object name>

Sending a message to an object:

         <object name> <message>


 Limitations in the Specification:

         o No Incest. A child class may not be a descendent of
           multiple classes which have an ancestor in common.

         o Instance variables may only be directly accessed in the
           class in which they occur. While they will exist in the
           child classes, they will not be directly accessible by
           the code in the child classes.
```

But inheritance is not all of object oriented programming. The most popular of the "pure" object oriented languages (as opposed to the features grafted onto existing languages) is Smalltalk-80. Key to Smalltalk is the idea that you do not act on an object, you send a message to an object and it then acts on itself. Earlier I discussed the problems of rigidly typed operators. This particular model, the idea of sending a message to an object, fits in very well with the Forth "noun verb" ideal. I can have code which says "DialogueWindow Open" and in another place says "OutputFile Open". In each case, I know what I mean, and because it is the object (either DialogueWindow or OutputFile) which decides the action, I can use the same "verb", preventing the name clashes or awkward syntax I might otherwise have to deal with. Other examples abound (I may want to append something to a file, and also append an object to a linked list), and in each case the common message allows the code to be

clear, while still being functional.

Drawbacks to Objects

Forth is by default an early binding language. That is, a function name inside of a definition binds itself to an action when the word is first compiled. If a lower level word is redefined, all other higher words which use it must be recompiled for the change to take effect. Because the action of the message is dependent on the object it refers to, these extensions tend to be late binding, the action is decided at run-time. This late binding has several advantages, one of which being the easy use of polymorphism (yet another theory word, it simply means the ability to redefine part of a class and have the change affect instances of that class which are already instantiated [in existence]). But it also costs us in the runtime efficiency of our code (it has to decide what to do each time the message is sent, rather than making that decision when the code is compiled). The use of objects does make code clearer, and it has definite advantages in terms of both programming style and ease of programming, but it costs us in speed. Whether or not this tradeoff is worthwhile is something which really must be considered on a case by cases basis.

Objects and Forth

This column (theory overdose and all) began as I became interested in object oriented programming and techniques. Rather than sit down and learn yet another language, I decided to take the theory, and implement an object oriented system in Forth. It is a tribute to the power of Forth that adding Smalltalk style Object-Orientation is not a terribly difficult task. The implementation syntax is described in Figure 4, and by the time you read this, the source code along with ports for various Forths should be freely available on various Forth networks (such as GEnie and Usenet's comp.lang.forth, as well as on the xCFB's of Forthnet). The idea was to blend objects with Forth, without losing the power of either. The first choice was to use the Smalltalk model. The second was to go with multiple inheritance. Here's why: assuming I have a class Lists, which implements the code for linked lists, and another class Files, which implements file handling. I can declare a class FileList with no additional code which will allow me to handle linked lists of files, just by declaring the class to be a descendent of both. It is this kind of modular approach which makes object oriented tools so powerful (again, it is possible to write atrocious code using object oriented extensions, but the extensions make it easier to write modular code if you so chose...there is no inherent protection against bad programming in any language).

How do You Implement This?

Warning this can get complicated!

Each class is a defining word (which makes the class structure itself a second order defining word, a defining word which defines other defining words). The structure of the class contains pointers to a linked list of methods (each method has a number, which is associated with a given message), a linked list of inherited classes, and a size field (other fields can be and will be added as this toolbox grows). Each object consists of a pointer to its class (which allows polymorphism, because changing the class will change already created objects, and also allows Forth code to check the classes of objects), and a block of allocated space in which the instance variables are stored. These objects themselves when executed leave a pointer to themselves on the stack (and you thought C went overboard with pointers). It is this pointer which a message uses to execute

Figure 5: A simple complex-number definition.

```
from object-extensions  import class as class
                        import end-class as end-class
                        import message as message
                        import var as var
                        import method: as method:
                        import end-method as end-method immediate
                        import subclass-of as subclass-of
end-imports

message add
message subtract
message store
message fetch
message display

class complex

        var real
        var imaginary
        variable complex-state  0 complex-state !

        method: add
                complex-state @ 0= if
                        real @ imaginary @
                        true complex-state !
                        rot add
                else
                        false complex-state !
                        imaginary @ +
                        swap real @ +
                        swap
                endif
        end-method

        method: subtract
                complex-state @ 0= if
                        real @ imaginary @
                        true complex-state !
                        rot subtract
                else
                        false complex-state !
                        imaginary @ swap -
                        real @ rot -
                        swap
                endif
        end-method

        method: store
                imaginary ! real !
        end-method

        method: fetch
                real @ imaginary @
        end-method

        method: display
                real @ .
                imaginary @ .
        end-method
end-class complex
```

the method, and because a pointer is the size of a Forth cell, it can be swap'd, dup'd or otherwise manipulated just as other values can (because the objects are meshed in with the parameter stack rather than being separated into an "object stack" it is not practical to add compile-time binding except in the simplest of cases). Instance variables (which are only used inside of a class definition) consist of an offset, which is added to an "instance base" which is computed at runtime to determine which position in the object corresponds to the instance variable. At runtime, a message uses its method number (a unique value) and searches the method list first of the object, and then of the objects ancestors, and executes the first method it finds which is coded to handle the given message. If no method is found, the deferred word NO-METHOD is executed (it is deferred to allow user definable error trapping.

Examples of Object Code

The package described above provides the basis for writing object oriented code, but what you do with it is up to the programmer. How accessible the innards of an object are to the outside world is dependent upon what messages the programmer decides to put in. If two objects are added (for

(Continued on page 8)

Improving the Ampro LB

Discard the 88Mb Hard Drive Limit

by Terry Pinto

Note from Jay Sage: When I met Terry Pinto in Portland in May, he told me about the technique he had developed to overcome the 88 Mbyte limit on the Ampro Little Board he uses to run his remote access computer system. My immediate response was, "Please write it up for TCJ!" And here it is.

That old saying that "Necessity Is The Mother Of Invention" certainly holds true. After purchasing my Ampro, I noticed that I was limited to a maximum of 88Mb of hard disk space. My feelings then were, WHO CARES! How could I ever fill that much space! I found a good price on a Priam V150 60Mb RLL certified hard disk and added it to the system. Well, you guessed it. It certainly didn't take long to fill up the disk. I run a BBS, Access Programming RAS, and my users were slowly eating away any free disk space I had. The time had come to enlarge the system. The problem was that I couldn't justify the expense of expanding to an 88Mb drive just to gain the additional space, and the extra 20Mb was a small addition that would disappear rapidly. The only solution was to engineer a way to utilize the fact that the Adaptec 4070 controller I used could support two hard disk drives. I purchased a copy of the source code to the Ampro BIOS and set to work writing HDS v1.00 (Hard Disk Select).

Before explaining how HDS works, a brief description of Ampro's drive table is in order. This table contains the information necessary to relate the logical drives to the physical drives on the system and to select the driver routine needed for each logical drive. The table is arranged in 16 groups of 4 bytes each for logical drives A-P. The information in byte 2 is used to select the physical drive on the controller (see Tables 1 and 4). Setting bits 7, 6, and 5 to a value of 000 will select physical drive 0 on the controller while a value of 001 will select physical drive 1.

Provided that both drives are physically compatible (for example, the same make and model) and are partitioned identically, simply changing bit 5 in byte 2 from 0 to 1 for a given logical drive (partition) will cause the system to access (read/write) the drive set up as SCSI drive 1 in place of the drive set up as SCSI drive 0. As you can see, the fact that both drives are partitioned the same is VERY IMPORTANT! The only thing I attempt to do with HDS is to tell the controller to use the other hard disk. HDS gives you double the disk space without the loss of any additional TPA.

Hard Disk Select

The Ampro BIOS can be set up to accommodate hard disk space up to 88Mb. This is due to the fact that a CP/M system can address sixteen drives, A-P, of up to a maximum of 8Mb of space each. Ampro defines drives A-D as floppy drives and drive E as the foreign format floppy. These allocations are predefined and cannot be changed, although by using the SWAP utility they can be moved around. For the purposes of this article, we will assume that the original drive mapping is intact and that you have not used SWAP to redefine the drive parameter tables. In fact, the use of SWAP is not restricted, and you may feel free to swap your drives at will either before or after running HDS.

With five drives being predefined, this leaves us eleven drives of 8Mb each, which gives us 88Mb of space. This will leave a TPA of about 56k. If you have elected to size your system to allow 88Mb of space, you may use HDS to 'mount' any eleven drives available to the system. What this means is that although you may have 22 logical drives attached to the system, you may only define eleven of them as currently active. This action is referred to as 'mounting' the drive. HDS will allow you to select any eleven drives available and make them active on the system. You may select an individual logical drive, a group of logical drives, or an entire physical drive. HDS will accept options passed on the command line describing what actions you want to take.

```
Syntax:  HDS [n|d]  (one option allowed - global select has
         priority)
       n=logical drive number (global select)
       d=list of drives to select
```

When HDS selects a drive from the appropriate hard disk, it toggles the selection. To reset the drive, you must either reselect the drive, or do a global select to drive 0. You should either select a global option or a set of individual drives. If both options are specified, the global select will occur.

```
Examples:
         HDS fghi  selects drives F,G,H and I from HD 1
         HDS bij  (the drives do not need to be consecutive)
         HDS c selects drive C from HD 1
         HDS 0 global select to HD 0 (reset).
         HDS   shows map of system
```

In the first example above, HDS will attach drives F,G,H and I from logical drive 1 to the system replacing their counterparts on drive 0. The drives do not have to be contiguous, as the second example shows. In this example, drives B, I, and J are attached, replacing their counterparts on drive 0. In the third example, just drive C is exchanged. In the fourth example, a global select is issued to select all drives on logical hard disk 0. This can be used as a quick method of resetting the system. The last example, HDS is run without a command line argument. This causes the system to display a map of the available drives on the system and how they are selected.

HDS will monitor both the QUIET flag and the WHEEL byte in order to control the display features and to provide system security. The use of the global select function requires that the wheel byte be set, if selected during assembly. If the wheel byte is not set, the user is given an illegal function error and control is returned to the operating system. To give you a higher level of security, I have provided a bitmap of the drives on the system. Setting any of the drives in this bitmap will secure the drive and will require that the wheel byte be set to mount that drive.

```
                                  ABCDEFGH IJKLMNOP
Example:  DRVMAP:    DB   01000000 00000000b
```

In the example above, drive B has been set as secure. This

```
Drive A    01 00 C6 00     Floppy Drives A-D
Drive B    01 11 C6 00
Drive C    01 22 86 00
Drive D    01 33 86 00
Drive E    02 44 00 FF     Foreign Format Drive E
Drive F    03 50 0A 01     Hard Disk Drives F-P
Drive G    03 60 0A 01
Drive H    03 70 0A 01
Drive I    03 80 0A 01
             |  |  |  |
             |  |  |  +--- SCSI Address
             |  |  +------ Drive Type Identifier
             |  +--------- Physical Offset and Drive Unit
Number
             +------------ Disk Drive ID  0 = Not Installed
```

Table 1. Physical Driver Table

```
          00 = Reserved for errors
          01 = Floppy Drive
          02 = Foreign Format Drive
          03 = Hard Disk Drive
          04 = Currently Undefined
          05 = Currently Undefined
          06 = Currently Undefined
          07 = RAM Disk (N/Systems)
```

Table 2. Values of the Disk Driver ID (byte 0)

```
   Bits 7654 Offset to base 0-F

   Bits 3210 Physical device address to be passed
                  to the respective driver.

   (For floppies, this is the drive unit#.  For
      hard disks, these are reserved.)
```

Table 3. Physical offset from start of disk parameter headers and the drive unit number (byte 1).

```
     Floppy Usage:
               Bit: 76543210
     Density         X         0=Single 1=Double
     Sides            X        0=Single 1=Double
     Sector#'s         X       0=Same 1=Contiguous
     Track Cnt          X      0=Down 1=Down Front - Up Back
     Alloc Unit          XX    00=1k  01=2k  10=4k  11=8k
     Sector Size           XX  00=128 01=256 10=512 11=1024

     Hard Disk Usage:
               Bit: 76543210
     LUN             XXX       Logical unit number (0-7)
     Reserved           X
     Alloc Unit          XX    00=1k  01=2k  10=4k  11=8k
     Sector Size           XX  00=128 01=256 10=512 11=1024
```

Table 4. Drive Type Identifier (byte 2).

```
               Bit: 76543210
SCSI address 0        X  This is the actual bit pattern
SCSI address 1       X   supplied during the SCSI select
SCSI address 2      X    routine.  No internal address
SCSI address 3     X     translation or bit scaling is done.
SCSI address 4    X
SCSI address 5   X
SCSI address 6  X
SCSI address 7 X
```

Table 5. SCSI bus address for hard disks (byte 3).

means that if the wheel byte is not set, the user could not issue a command of 'HDS B'. Attempting it would generate the illegal function error and return the user to the operating system.

For use on a remote access system, I've tied the display functions of HDS to the QUIET flag. When the quiet flag is on, no screen output is generated during the selection of drives. This will enable you to include HDS in an alias and have it totally transparent to the user. With the QUIET flag off, a signon message will appear each time you run HDS. The same is true for the display function of HDS. If you call HDS without a command line option, it will give you a map of what drives are selected and from which physical drive. Table 6 shows an example of a typical run of HDS.

```
A0:BASE>HDS
Hard Disk Select v1.03 (c) 05/17/89
Written by Terry Pinto
ZCPR v33   ZRDOS v19   Ampro BIOS v38   PHYTAB D072H
(Type 3 - Loaded at 8000H)
Selected Drives

A - Floppy Drive
B - Floppy Drive
C - Floppy Drive (not installed)
D - Floppy Drive (not installed)
E - Foreign Format Drive
F - Drive 0
G - Drive 0
H - Drive 0
I - Drive 0
J - Drive 0
K - Drive 0
L - Drive 0
M - Drive 0
N - RAM Disk
O - Does Not Exist
P - Does Not Exist
```

Table 6. Example of screen display after running HDS.

```
A0:BASE>HDS FGHI
Hard Disk Select v1.03 (c) 05/17/89
Written by Terry Pinto
ZCPR v33   ZRDOS v19   Ampro BIOS v38   PHYTAB D072H
(Type 3 - Loaded at 8000H)
Selected Drives

A - Floppy Drive
B - Floppy Drive
C - Floppy Drive (not installed)
D - Floppy Drive (not installed)
E - Foreign Format Drive
F - Drive 1
G - Drive 1
H - Drive 1
I - Drive 1
J - Drive 0
K - Drive 0
L - Drive 0
M - Drive 0
N - RAM Disk
O - Does Not Exist
P - Does Not Exist
```

Table 7. Example of screen display after running HDS a second time.

You can see from the output that HDS reads the parameter table to determine if a drive is on the system and, if so, what kind of drive it is. Notice that all four floppy drives are always defined by Ampro but that on this system drives C and D are 'not installed'. The foreign format floppy is drive E and is used for the reading/writing of disks formatted for other systems. The display shows that hard disk drives F-M are defined and are currently the drives residing on the hard disk defined on your system as being physical drive 0. Drive N is defined as a RAM Disk (N/Systems) and drives O and P do not exist on the system.

If you select drives FGH and I, you get the output shown in Table 7. Notice that the drive identifier for drives F,G,H and I now show that these drives are currently attached from the hard disk defined on your system as physical drive 1.

After the selection of the requested drives, it is necessary for HDS to perform a disk reset to log in the newly defined drives. To accomplish this, HDS must know what type of DOS you are using. A check is done to determine if you are running vanilla CP/M 2.2, ZRDOS, or ZSDOS. In either case, an appropriate disk reset is called upon to log in the new drives. During the check process, HDS also determines if you are running a vanilla CP/M or a ZCPR3 system and gets the version number. This is necessary to enable HDS to find the physical tables in your BIOS. The type of system you are running will be displayed during the signon if the QUIET flag is off. For now, HDS only detects Z33, Z34,

```
HDSEL echo;The Following Hard Disk Selections are Available;echo;echo
      AMPRO;echo;MSDOS;echo;ZCPR;echo;SYSTEM;if wh;echo;EXTEND;fi;echo
AMPRO  echo;Loading AMPRO Drivers - Please Wait...;hds 0;hds c
MSDOS  echo;Loading MSDOS Drivers - Please Wait...;hds 0;hds dfghi
ZCPR   echo;Loading ZCPR Drivers - Please Wait...;hds 0
SYSTEM echo;Loading SYSTEM Drivers - Please Wait...;hds 0
EXTEND echo;Loading EXTENDED Drivers - Please Wait...;hds 1
```

Figure 1: ALISA.CMD file.

and CP/M operating systems. Future versions may include detection of CP/M plus.

Online examples from Access Programming...

To enable an easier way for the remote user to use HDS, I've placed the alias' shown in Figure 1 in my ALIAS.CMD file. (The case switching symbols have been eliminated to make the alias a little easier to read).

The HDSEL command will display the following selections:

```
A0:RCPM>HDSEL

The Following Hard Disk Selections are Available

AMPRO
MSDOS
ZCPR
SYSTEM
EXTEND

A0:RCPM>
```

This gives my users an online guide to what is available. Running AMPRO will attach drive C from HD 1. This is where all the Ampro specific files are located on my system. MSDOS will load drives D, F, G, H and I from HD 1 and allow the users to access the MSDOS programs stored there. ZCPR and SYSTEM actually perform the same function for now. ZCPR will grant access to the ZCPR specific files on HD 0. The SYSTEM command will serve as a system reset. I've made BYE an alias on my system allowing me to include the command 'HDS 0' in the signoff module. This effectively resets my system for the next caller. This allows me to maintain a default status so that each new caller will enter the operating system with the same drives attached. The EXTEND command selects all drives from HD 1 to the system. Much of my commercial software is kept here so I've set HDS to require that the wheel byte be set to do a global select to drive 1. I've also set up drive B as a secure drive. This is done by setting the corresponding bit in the drive bitmap to 1. If HDS detects the security bit on any drive passed on the command line, access will be denied.

Acknowledgments

Many hours of research have gone into the development of HDS. Release of this software would not have been possible without the information that was obtained from the source code of version 3.8 of the Ampro BIOS. Much of the information describing the bit selections necessary for HDS were quoted directly from the source code listing.

If you would like a copy of HDS or have any questions about the software or its operation, please contact:

Terry Pinto
Access Programming RAS
14385 SW Walker Rd. B3
Beaverton, OR 97006

(503) 646-4937 6:00pm-10:00pm PST
(503) 644-0900 300/1200/2400 8N1
PCP ORPOR StarLink 9164/222 (local exchange)

Forth Column

(Continued from page 5)

example, complex numbers), should the sum be put in one of them, in a temporarily created object, in a static but previously defined object, or should the result be left on the stack for the programmer to deal with? These answers depend more on how much of a "pure" system you want to use. (Smalltalk systems generate and destroy an incredible number of temporary objects each second). Figure 5 is a simple complex number definition which handles only addition and subtraction, as well as the displaying of the object. By using a variable which is common to all members of the class, it is possible for objects to pass messages to each other (such as in this case what state an operation is in). Furthermore, the word SELF allows an object to send itself messages (letting an object use code defined in its ancestors).

An End to Objects

Looking back over what I've written I see an awful lot of theory in this column. Whether or not this is good or bad, I don't know. This is a new column, and I don't know whether or not you want more articles like last issue (all implementation details), or this one (a large amount of theory). Please tell me what you want to read. For that matter do you want language extensions, language tutorials, or examples of Forth applications?

Right now Forth is going through a standardization process, the directions the language will take in the future are to a large degree being decided now. What form the standard takes, and whether or not the upcoming ANS standard is usable for Forth programmers depends on the actions of Forth programmers now (inaction now could well lead to much grumbling over beer later). So (he said jumping firmly onto the soapbox) let the people on the Standards committee know what you want, think, or absolutely demand in Forth (they do want to know). The committee can be contacted via GEnie (there is a large discussion on CATegory 10 of the GEnie Forth RoundTable about the ANS standard), or via comp.lang.forth on Usenet (and if you can't get to either I will be glad to forward any comments, questions, or threats to them). Also coming up soon is the second annual SIGForth Forth conference (in Dallas in February). Considering that this year, despite problems with the publication of the Call for Papers and scheduling conflicts, the first conference still produced more papers than attendees and some truly breathtaking ideas ("You are going to write this up for the proceedings!" kept popping up more and more often after the informal talks). These conferences are perhaps the best way to keep up with what is happening in Forth, the discussions with other Forth programmers alone make them worthwhile. •

Data Structures in Forth

by Joe and Marla Bartel, Hawthorne Technology

Is Forth Out of Date?

In the early days of computing, programming languages didn't provide for complex data structures. Languages like BASIC and FORTRAN only have arrays and single variables. Personal computers have greatly affected programming style. COBOL has always had complex data structures but only a few people use COBOL on a personal computer. It was only when Pascal and C became popular that many people started using a language that supported complex data structures. It was then that they started to think about using fields and records. Forth is one of those languages that has been around since the early days of computing. Its definition does not include data structures. Does this mean that Forth can not be the best language choice for an application that should be handled with records? This article shows why Forth shouldn't be neglected when dealing with data structures.

Are Pascal and C the Only Way to Go? (Data Structures Make the Difference.)

In Forth it is necessary to be explicit about pointer usage. Languages like C or Pascal do the same things with pointers but the means are hidden from the programmer. One result of this is that in C or Pascal, large, slow access routines can be generated without the programmer being aware of it. There is nothing that can be done in C that can't be done just as well in Forth when field words have been added. Another advantage to using Forth is that it doesn't take 500K of RAM and a hard disk to do useful program development.

Some programming styles and languages fit better with a particular programming problem than others. There are many cases where data structures can organize data and make a program easier to understand and cleaner to write. Pascal and C have data structures. Unfortunately there are many times, particularly in embedded systems for using small micros, where Pascal or C is not available, or costs too much, or is too inefficient. By using the same types of structures in Forth, you can continue to use familiar structures and algorithms. By doing so you can avoid the problems that might creep up on you when you are forced to use new methods.

Forth is Dynamic so Data Sstructures (Records / Field Words) Can be Added

Forth is a very flexible language that is easy to modify. A very useful way to take advantage of Forth's flexibility is by adding field words. With field words it is easy to create any kind of record or structure you need just as you can with Pascal or C.

HTForth (Hawthorne Technology) has field words included to simplify the use of abstract structures. The field words define an address that is an offset from a base address. The base address can be a fixed location in memory or it can be a pointer that is on the stack. The definitions for field words are provided in Forth so they can be used with any Forth. The words that are used for fields in HTForth are written in assembler and generate macros that are inserted in-line in the code when they are used. The result is the same as with threaded Forth but the speed is greatly improved. The examples presented here should work with any Forth but they have not been extensively tested.

HTForth field words:

```
BFIELD   defines a 1 byte field
WFIELD  defines a 2 byte field
LFIELD  defines a 4 byte field
$FIELD  defines a variable length field
```

You can achieve the same effect of the other field words by using a size of 1, 2, or 4 with the $FIELD.

These are defining words that create new words that have special actions when used. The compile time behavior is to create a new word that will add the value that was on the stack at compile time to the value that is on the stack at run time. The other compile time action is to increment the value on the parameter stack by the size of the field that is being defined.

The declaration of a record using the field words is like the declaration of a record type. No actual memory is allocated for any variable but the offsets and order of the fields is set up for when they are used. To actually allocate memory for a record that is defined some other method must be used.

The listing of the Forth definition for the field words can be used to define field words for any implementation of Forth that does not include them. Data structures can then be created with the field words that have been defined.

How to Define and Use Records and Pointers

The main example we will use will be a symbol table for an assembler. The symbol table for a conventional assembler has an entry for each symbol the user defines. Each entry consists of several fields that must be kept together for each table entry. This group of fields would be a record that you would define in HTForth like this:

```
    0                    (  offset of first field       )
    LFIELD   LINK        (  4 byte link to next symbol  )
    WFIELD   TAG         (  2 byte tag                  )
    LFIELD   SYMVAL      (  4 byte value                )
 12 $FIELD   SYMNAME     ( 12 byte symbol name          )
    CONSTANT SYMREC      (  size of record              )
```

The field words use the top of the stack as the offset into the structure for the field currently being defined. The size of the field being defined is then added to the top item on the stack. This way a field can be inserted into a structure and all the offsets used will be automatically adjusted.

After all the fields in a record have been defined, the value on the stack can be made into a constant that will give the size of the whole structure. A constant that has the size of the structure is very useful for defining arrays of structures, allocating new instances of a structure, and for reading or writing disk files. The constant can also be used to increment or decrement a pointer into an array of the structure. By using a named constant that is produced when the structure is defined you don't have to worry about the structures' exact size and future changes will be much easier to make.

Here are some examples of how to access the elements of records:

To access the tag of the symbol entry that variable SYMPNT points to, use:

```
SYMPNT @ TAG @W
```

To allocate a fixed array of 30 symbols use:

```
30 CREATE SYMTAB SYMREC 30 * ALLOT
```

To access the value of the 5th entry of the array use:

```
SYMREC 4 * SYMTAB + SYMVAL @
```

To allocate a new data item of the type SYMREC:

```
HERE SYMREC ALLOT
```

The result is the address of a new node that then can be linked into the rest of the table.

How to Define and Use Substructures

Substructures are records that are one element of another record.

A Pascal example of this would be:

```
friends = record
          name    : string [ 32 ] ;
          address = record
              street        : string [ 32 ] ;
              city          : string [ 16 ] ;
              state         : string [ 2 ] ;
              zip           : string [ 5 ] ;
              end ;
          phone             : string [ 15 ] ;
          notes = record
             birthday       : date ;
             favorite_color : string [ 10 ] ;
             hobbies        : string [ 20 ] ;
             end ;
         end ;
```

In this example, NOTES is a substructure of FRIENDS. You could go on to another layer of substructure by making HOBBIES a record.

To define substructures, push a zero on the stack and define the fields that make up the substructure. At the end of the definition of the fields use the variable field word $FIELD followed by a name (in this case 'NOTES'), to give the substructure a name. This will make the size of NOTES equal to the total size of the substructures' fields.

A substructure that will be common to many structures can be defined separately and its size saved in a named constant. The size value that is saved can be used with $FIELD to create a name and space for the substructure in the new structure being defined. This way a common substructure doesn't have to be re-defined for each of the structures it is to be used with.

```
   0
32 $FIELD NAME
   0
   32 $FIELD STREET
   16 $FIELD CITY
    2 $FIELD STATE
    5 $FIELD ZIP
$FIELD ADDRESS
32 $FIELD PHONE
   0
   WFIELD BIRTHDAY
   BFIELD SEX
   16 $FIELD HOBBIES
$FIELD NOTES
CONSTANT FRIENDS
```

To create a record of this type for BILL and one for ART we can write:

```
CREATE BILL FRIENDS ALLOT
CREATE ART  FRIENDS ALLOT
```

To access the ZIP code for BILL we would write:

```
BILL ADDRESS ZIP
```

And to do the same for ART we write:

```
ART ADDRESS ZIP
```

The parameter stack would then have the address of the first byte of his zip code.

How to Define and Use Variant Records

Variant records, or unions (as they are sometimes called), are when two or more fields are assigned to the same location in a record. A tag field is often used to determine which of the variant records is stored in the structure. For example, a record for DINNER might have a tag to indicate if you are eating out or cooking at home. If the TAG indicates EATING OUT, the RESTAURANT LIST will be using the WHERE TO GO field in your record. If the tag indicates COOKING AT HOME, the GROCERY LIST will be stored in that field. The programmer has to keep track of the tag indication so you don't end up at the local steak house and order a dozen eggs and gallon of milk.

Figure 1:Cross assembler look up routine in Pascal

```
{ ------------------------------------------------------- }
{ Pascal linear search of alphabetical symbol table list -- }
type
  string12       = string[ 12 ];
  symbolpointer  = ^symrec;
  symrec         = record
                     link    : symbolpointer;
                     tag     : integer;
                     value   : integer;
                     symbol  : string12;
                     end;
{ Global Variables }
var
  counter            : integer ;
  firstsym, cursym   : symbolpointer ;
{ ------------------------------------------------------- }
function newsym( labnam : string12 ): symbolpointer;
var
  tempsym  : symbolpointer;
begin
  new(tempsym);
  newsym := tempsym;
  newsym^.link   := nil;
  newsym^.tag    := 0;
  newsym^.value  := 0;
  newsym^.symbol := labnam;
  counter := counter + 1;
end;  { newsym }
{ ------------------------------------------------------- }
procedure lookup( var labnam : string12; var whr : symbolpointer );
var
  nextsym                      : symbolpointer;
begin
  nextsym := firstsym;
  if firstsym <> nil then begin
    whr := nil;
    while whr = nil do begin
      cursym := nextsym;
      nextsym := nextsym^.link;
      if cursym^.symbol = labnam then whr := cursym
      else if labnam < cursym^.symbol then begin
        firstsym := newsym(labnam);
        whr := firstsym;
        firstsym^.link := cursym;
      end
      else if nextsym = nil then begin
        cursym^.link := newsym(labnam);
        whr := cursym^.link;
      end
      else if labnam < nextsym^.symbol then begin
        cursym^.link := newsym(labnam);
        whr := cursym^.link;
        cursym := cursym^.link;
        cursym^.link := nextsym;
      end;
    end;
  end
  else begin
    firstsym := newsym(labnam);
    whr := firstsym;
  end;
end;  { lookup }
{ ------------------------------------------------------- }
```

Figure 2: Cross assembler look up routine in Forth

```
( ------------------------------------------------------ )
( ---------- Field words for F-83 style Forth ---------- )
: bfield                ( n - n+1 )   ( byte field )
    dup 1+ swap
    create  ,   does> @ +
    ;
: wfield                ( n - n+2 )   ( word field )
    dup 2+ swap
    create  ,   does> @ +
    ;
: lfield                ( n - n+4 )   ( long field )
    dup 4+ swap
    create  ,   does> @ +
    ;
: $field                ( n s - n+s )   ( variable field )
    over + swap
    create  ,   does> @ +
    ;

( ------------------------------------------------------ )
( -- FORTH linear search of alphabetical symbol table -- )
( ------ Define a symbol table record ------------------ )
    0
    1field      link        ( pointer to next symbol )
    wfield      tag         ( 2 byte tag )
    1field      value       ( 4 byte value )
    12 $field   symbol      ( 12 byte symbol name )
    constant symrec     ( size of record )

    0 constant nil
    variable counter   variable firstsym
( ------------------------------------------------------ )
: $<12            ( ^stringA ^stringB -- tf )
  12  0  do
    over C@  over C@          ( -- ^A ^B a b )
    <> if  leave  then
    1+ swap  1+ swap
  loop
  C@ swap  C@ swap  <
  ;
( ------------------------------------------------------ )
: $=12            ( ^stringA ^stringB -- tf )
  12  0  do
    over C@  over C@          ( -- ^A ^B a b )
    <> if  leave  then
    1+ swap  1+ swap
  loop
  C@ swap  C@ =
  ;
( ------------------------------------------------------ )
: newsym     ( create a new symbol )
  here symrec allot
  nil over link !
  0  over tag !
  0  over value !
  over over symbol 12 cmove
  swap drop  counter 1+!
  ;
( ------------------------------------------------------ )
variable nextsym  variable labnam  variable whr
variable cursym

: lookup  ( labnam -- whr )
  labnam !   nil whr !
  firstsym @ nextsym !
  firstsym @ nil <>
  if
    begin  whr @  nil =  while
      nextsym @  cursym !
      nextsym @ link @  nextsym !
      cursym @ symbol  labnam @  $=12
      if
        cursym @  whr !
      else labnam @  cursym @ symbol  $<12
        if
          labnam @  newsym  dup  firstsym ! whr !
          cursym @  firstsym @ link !
      else  nextsym @  nil =
        if
          labnam @ newsym  dup  cursym @ link  ! whr !
      else  labnam @  nextsym @ symbol  $<12
        if
          labnam @  newsym  dup  cursym @ link !
          dup  whr !  cursym !
          nextsym @  cursym @ link !
      then then then then
    repeat
  else
    labnam @  newsym  dup  firstsym ! whr !
  then
  whr
  ;  ( lookup )
( ------------------------------------------------------ )
;S
```

To define variant records DUP the size constant value on the stack and then define the first variant of the record. The variant can have multiple fields defined and can have variants of its own. After defining the variant portion use MAX to get the larger size of the structure. This will insure that there is enough space allocated for the largest variant when variants are of different sizes. This method can also be used to create alternate definitions on the same level. If two different variants are wanted as substructures use MAX before + to add the largest variant substructure to the main structure.

Sample Forth Data Structure Code

The program examples in Figures 1 and 2 are from a lookup routine for a cross assembler Marla wrote this summer. To make it more understandable to those who are new to Forth, the same routine is presented in Turbo Pascal and then in F-83 style Forth. The cross assembler was written in Pascal. We carefully translated the Pascal into Forth but have not actually run the Forth code. The data structures allow for an almost line for line translation from Turbo Pascal to Forth but the Forth version results in more compact code.

When a program is being assembled, as each symbol is encountered, the lookup routine is called and the table is searched. If the symbol is not found, it is added to the table in alphabetical order. These symbols are being stored in a singly linked list. The link field of the defined record points to the next record in the list. The lookup routine steps through the list comparing the symbol from the assembly to those stored in the list. If it finds a match for the symbol, it returns a pointer to the record containing the matched symbol. The cross assembler can then use the information in the record or add things such as flagging multiple definitions of labels, undefined symbols, or recording the value of the symbol to be used in the rest of the program.

Each time the lookup routine compares a symbol that doesn't match, it checks to see if the entry in the list is still higher in the alphabet than the symbol being searched for. If it is not, you know the symbol is not in the list because you have passed the place where it would have been. If the symbol is not found, the NEWSYM routine is used to make a new entry in the table. The new entry is then linked in to the list just before the last item it was tested against.

List before new symbol is inserted:

```
Asymbol.link → Bsymbol.link → Csymbol.link → Esymbol.link
```

List after new symbol is inserted:

```
Asymbol.link → Bsymbol.link → Csymbol.link      Esymbol.link
                                        ↓       ↑
                                      Dsymbol.link
```

By linking things into the list in this manner, the symbols are kept in alphabetical order. This has several benefits. You don't have to

(Continued on page 26)

Advanced CP/M

Hacker's Death Greatly Exaggerated! and Z-System Command Scheduler

by Bridger Mitchell

In a column closing out the fifth anniversary issue of *Computer Language* (July, 1989), Tim Parker laments the "Death of the Hacker." He recalls that in the early days, clever programmers pushed CP/M's limits by adding capabilities to the operating system, and were driven by the desire to share innovations and discoveries with fellow programmers. But now, he writes, the evolutionary process is disappearing. "CP/M reached its zenith with ZCPR just as the IBM PC emerged...The hacker is a dying breed: there is little left to hack."

Surely, I wrote the editor, Mark Twain would say that the report of the "Death of the Hacker" is greatly exaggerated! Indeed, CP/M continues to attract exceptionally talented and community-minded programmers who have steadily pushed the frontier out far beyond the capabilities of the first PCs.

A Celebration of System Hacking

CP/M from its beginnings has been a relatively portable and open operating system. Gary Kildall concocted its first versions to enable early software to run on 8080 computers that used different types of peripheral hardware—in those days paper tapes, cassette tapes, eight inch floppy disks, and teletype terminals. A fundamental design concept was to encapsulate the "low level" hardware interface code into one BIOS module, put the file management code into the BDOS module, and leave command interpretation in a third CCP module.

CP/M system hackers—the early 8080 computer hobbyists—got a copy of the (legendarily unreadable) Digital Research Inc. (DRI) manuals and CP/M 1.4 files and set to work writing a BIOS for their own hardware. As microcomputers spread, small business applications flourished; each original equipment manufacturer (OEM) wrote a BIOS for its particular hardware and bundled a bootable CP/M 2.2 system with the equipment. In this way CP/M spread to a very wide variety of hardware platforms.

The ready availability of a running CP/M system shifted hackers' attention from getting a BIOS running to improving the performance of the user's interface with CP/M -- the command processor. This lode has been a rich one, indeed, and it has been mined continually since the first ZCPR committee replaced the DRI 8080 CCP with a Z80 version and packed additional features into the 2K module.

Today's Z-System is a mature, many-featured descendent of those early system hacks, and one that continues to evolve. The system provides a path of directories to search for a command, conditional execution of commands (the Flow Control Package), command history-and-recall (EASE, LSH, BGHIST), error-processing (EASE, ZERRLSH, BGERR, ERRORn), named directories, and password security. Command shells can provide a friendly visual menu environment tailor-made to the user's types of applications (ZFILER, MENU, VMENU, FMANAGER).

Message buffers allow applications to communicate results to their successors. Command-line and even *user input to programs* can be automated (ZEX).

Aliases give the user great power. In their simplest guise they substitute a single, readily remembered word for a complex set of command instructions (SALIAS). But the extended command processor ARUNZ has extensive power to symbolically manipulate the user's command line. Parameters return not only whole tokens, as in MS-DOS, but can decompose the tokens into their components (drive, user, filename, filetype). Other system information is also available (date and time, memory and register contents, system file names, etc.).

In the Z-System, extended command processing is automatic. Thus, the user can extend—in any way he wishes—the entire command processing system. When a user's command cannot be processed in its present form by the Z-System command processor, it is automatically handed over to another (possibly far more powerful) command processor. ARUNZ, with its great power and flexibility, is rightly viewed by many users as the single most important Z-System tool.

Perhaps the ultimate Z-System hack has been to make all of these features effectively self-installing. Until the arrival of NZCOM and Z3PLUS (and their named-common relocatable ZRL files) the many ZCPR advances still required BIOS surgery and reassembly. No more—the Z-System can be installed from menu-driven instructions on virtually every Z80 computer.

BackGrounder ii, another marvelous CP/M system advance that surpasses such MS-DOS system accessories as SideKick, enables users to switch between two active applications, and to use an extended set of built-in commands while a program is active. It provides two full-screen windows where hardware permits, and the ability to "cut" information from one task and "paste" it into a second.

A few CP/M veterans have pried open the BDOS module, diagnosed its few, obscure bugs, and then gone on to code improved versions. This is system hacking at its most demanding, for a single oversight can bring entire filesystems to total destruction. ZSDOS is the latest, best tested, and most accomplished of these efforts. It includes generalized search-path features and the ability to make files publicly accessible from any directory.

---

*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, or at (213)-393-6105 (evenings).*

Another noteworthy system hack is in the crossover category. Several application tools provide the ability to transfer files between MS-DOS and CP/M disk formats (MediaMaster, UniForm). The most ingenious of these is DosDisk, which effectively converts a drive to MS-DOS and allows CP/M programs to use and modify files on a DOS disk directly. In addition, DosDisk supports the MS-DOS hierarchical subdirectories. And, it preserves file date stamps between MS-DOS and CP/M systems.

DateStamping Matures

Which brings me to the growing availability of file time-and-date stamping. It's been gratifying to watch DateStamping spread to more and more CP/M systems since the "early days" when Derek McKay and I came up with a design to overcome this basic deficiency in the original CP/M 2.2 operating system. We strove for maximum portability, with the aim that *every* CP/M 2.2 system could be extended to include automatic filestamping, and users could exchange disks between systems and maintain accurate file stamps.

The design has withstood the test of time. The original DateStamping system runs on both 8080 and Z80 systems around the world. The resident stamping module can be loaded as a resident system extension (RSX) below the command processor, or as code installed above the BIOS in highest memory. We released DateStamper with two fundamental tools -- SDD, a version of Super Directory, that displays file datestamps, and DatSweep, a multi-purpose file utility that also displays and selects files by datestamp.

Such outboard DateStamping works very well, but it was always an appendage. About two years ago, two efforts to integrate file stamping directly into the operating system began. Hal Bower and Cam Cotrill, both users of DateStamper, started building DateStamper into their formative ZSDOS. Independently, Carson Wilson was experimenting with a filestamping format similar to that used by CP/M Plus in his prototype Z80DOS tests.

Carson and I had a vigorous and interesting exchange of views on the Lillipute Z-Node in which we debated the merits of filestamp formats and the value of a unified approach. Finally, out of all of this emerged a united and technically unsurpassable team—Bower, Cotrill, and Wilson—working to develop a single new CP/M 2.2 DOS. Integrated DateStamping is now a reality in the new ZSDOS operating system.

DateStamping Tools

As DateStamping has become more widely used, the number and variety of tools that use its features has expanded steadily. File datestamps can be maintained when files are compressed and later expanded (CRUNCH and UNCRUNCH v 2.3D), archived and extracted from libraries (LPUT, LBREXT), and copied (COPY, DATSWEEP). This crucial functionality "marries" a file and its date of creation and modification; as a result you always know which is the latest version of a manuscript, program or database, even when it has been copied, compressed, and put into deep storage in a library on another system.

Disk directories can be viewed and sorted by date and time (FILEDATE, ZXD, SDD).

We have tools for manipulating files based on temporal relations. CRUNCH and DATSWEEP can process files by date. And MAKE can determine whether interdependent files are out of date and automatically compile, assemble, link, or otherwise process them into updated condition. (MAKE deserves a column of its own; I hope to do one before much longer.) Jay Sage has suggested the need for a small, general-purpose tool, say DATECOMP, that would compare or test two files' datespecs and set the error flag or a register according to the result of the test.

As I mentioned earlier, DosDisk provides a unique link between the CP/M and MS-DOS operating systems by preserving datestamps when files are copied to or used in MS-DOS format.

Quite appropriately, all of these applications use the filestamp feature of DateStamper. But DateStamper has another, equally portable feature that is more direct—a hardware-independent method of obtaining the current date and time. DateStamper's portable access to the system clock is used by ZCAL, by MEX Plus overlays, and a number of tools that incidentally display the date and time in the course of their major tasks.

Z-System Command Scheduler

Reading "Death of the Hacker" and reviewing the progress of CP/M led me to muse about what we *haven't* accomplished with system extensions. One of them is an alarm-clock feature—the ability to set appointment reminders or schedule tasks for automatic execution.

Until recently I had dismissed this as not really feasible for CP/M. After all, "smart" pop-up calendars in MS-DOS systems hook into the interrupt-driven clock, and job-schedulers in minicomputers spawn independent processes under multi-tasking operating systems.

But gradually it dawned on me that we *do* have the essential system ingredients for a scheduler! DateStamper provides the access to the system clock and the Z-System multiple-command line provides a queue for future tasks—there must be a way to use these components to start a task at some future time! I kept mulling over these facts, trying to see how they could be made to fit together. It began to take shape ... suppose we squirrel a command away in protected memory and somehow monitor the clock regularly until the set time arrives. When it does, we move the command into the multiple command line. After that the ZCPR command processor will execute it when the current task is completed.

This was getting exciting! But how to rig things up to monitor the clock while everything else is happening on the system? Unlike MS-DOS, CP/M has no BIOS software interrupt (or BIOS jump) that provides regular clock ticks. My first thought was to design an interrupt-service routine that could extend the host system's real-time clock routine. This would be complicated, and system-specific, and probably introduce unknown problems.

At this point Cam Cotrill reminded me that the trick of monitoring the BIOS console status routine, which BackGrounder ii uses so successfully, should work here too. (I'm good at forgetting my best ideas!) And, indeed it does. We can intercept calls to constat, call the DateStamper clock and compare the current time with the alarm time, and finally proceed to check console status. It will work because the DateStamper clock is implemented by low-level code that does not endanger BIOS or BDOS routines that happen to be in use at that moment.

As this concept took clearer form I rushed to hack a proof-of-concept test. I had been cleaning up the development work for ZEX 5.0 and thus had the routines for hooking up an RSX and pushing a command onto the Z34 multiple command line readily available. In one evening I succeeded in lashing together a prototype "AT". This scrawny, still primitive beast with wires poking out everywhere and dozens of stub ends, was silently monitoring the DateStamper clock from its position in high memory and ready to pop into action at a hard-wired, preset time. The "hour" arrived, and, lo, the command line changed. And, when I exited the debugger, my "ECHO TESTING" command ran!

AT Takes Shape

"The rest is history." Except it's never that easy! At this writing, I'm steadily teaching the AT puppy (named for her UNIX granddad) some manners, though she's not yet fully housebroken. I am again mildly surprised at how much code, and repeated testing, goes into protecting a program against both user mistakes and system errors (not to mention the programmer's own blunders!). My first prototypes are very raw, user-intolerant beasts, demanding precise operator attention to prevent horrendous crashes. They often require, and assume, all of the operating system features I know are present on my test machine. Slowly, as the central functions and data structures take more solid form, I add increased error checking, recovery, and finally user-interface features.

Toward the end of this phase I will again seek out a diverse group of "alpha testers"—stalwart souls whose joie de vivre encourages them to live recklessly with infested and untried code. The ideal hacker for this role is doggedly persistent in trying to reproduce an anomaly, suspicious of "obvious" explanations for aberrant performance, observant of the precise sequence of events, patient to a fault in accepting yet another test version from the hapless developer, suffers wordlessly when his hard disk is crashed, and doesn't complain when his suggestions for improvements are brushed aside! Moreover, he routinely uses all sorts of obscure hardware, runs the widest spectrum of software applications, and has superb diagnostic tools at the ready. Finally, after pinpointing the precise cause of the developer's mistake, he cuts through the fog with a clairvoyant suggestion that I could perhaps more easily "do it *this* way", lays out a brilliant simplification, and then modestly declines to take credit for his insight!

This breed of hacker is alive and well. Most computer users have heard of beta testers, but it is the *alpha* testers who are the unsung test pilots of our field. I owe much of the success and quality of my CP/M programs to them, and I salute their cockpit courage and good humor -- Jay Sage, Howard Goldstein, Cam Cotrill, Hal Bower, Rob Friefeld, Al Hawley, Bruce Morgen.

At this writing, AT has just been submitted to the alpha circle, and is moving ahead. A beta version might be found on Z-Nodes about the time this TCJ reaches you. I am particularly interested in feedback and suggestions from all types of users. A scheduler is, by its nature, a tool of especially wide possibilities, and I expect several of you will invent some quite unexpected uses for it.

AT's commands

At the moment, this is how it will work. You type:

```
AT  <date> <time> command_line
```

AT then adds that command line to its internal database. If there is no earlier command in the database, it schedules that command by informing the AT RSX of the specified alarm time. Then, when that time arrives, the RSX causes AT to run again; this time it fetches the command line from the database and pushes that line onto the command processor's multiple command line. Finally, it also schedules the next command in the data base. Immediately thereafter, your command runs.

So far I have largely resisted the temptation to make AT any sort of command *processor*. I think it's best to maintain the clean division of labor between scheduling (AT.COM), alarm sensing (AT.RSX), and command processing (ZCPR34). ARUNZ's extended command processing facilities, and ZEX scripts, enable you to expand the scheduled command line into extensive and arbitrarily complex sets of tasks.

I am experimenting with one variant command:

```
AT <time> RING command_line
```

which rings a set number of times as soon as the alarm time is reached, and then executes the command line whenever the current application has finished.

AT will eventually include the ability to schedule periodic commands (daily, weekly) and a basic set of options for maintaining and displaying the database of scheduled commands.

Here are a few initial ideas for commands.

```
AT 08:00 DATEBOOK -- review daily calendar
AT 17:30 RING ECHO CALL JAY
AT 21:00 MEX READ DOWNLOAD LADERA
AT 23:50 BACKUP A:
```

I hope the promise of a command scheduler will prompt several of you to develop new applications. And I am eager to learn what uses all readers may find for AT.

A TEX Postscript

I was just putting this column into the mail to our editor when word of another, earlier command scheduler effort reached me. Called TEX (Timed Execution), it's by Ron Murray. The discovery of a new, kindred Z-System development from down under (Ron is in Perth, Western Australia and can be reached via Z-Node 62 there), one that took more than a year to migrate to me in southern California, is somehow slightly akin to the Voyager 2 transmissions from Neptune that have so fascinated us in the last weeks!

I've just begun to look at TEX (not to be confused with the powerful formatting language TeX). It is somewhat like AT, but also quite different. TEX is a Z-System shell. It runs as an application in low memory whenever the system command line is empty, and accepts normal command lines from the user. It also accepts scheduling commands, which TEX stores in a separate database file. Once a command has been entered in the database, TEX checks the clock against the next-scheduled command, and when its time has come, it pushes the command onto the multiple command line for execution. When the command line buffer becomes empty, the command processor again invokes TEX (because it is a shell) and it resumes checking the clock and getting the next user command.

Ron's original version was based on Carson Wilson's first BDOS replacement, Z80DOS. Jim Lill has taken Ron's modular code and, by using the now-standard routines in DSLIB, has been able to make TEX version 1.3 portable across all DateStamper, ZSDOS, and CP/M Plus Z-Systems. This is a particularly apt demonstration of the way the Z-System can grow and spread across diverse CP/M platforms.

TEX and AT will surely learn from each other in the coming months!

TEX's implementation as a shell allows the TEX scheduler to be added to the system without permanently reducing memory available to applications (the TPA). TEX runs in low memory, and this effectively removes the handy GO command from ZCPR's repertoire. (Although TEX could be relinked to run as a type-3 or type-4 tool in upper memory, it is large enough that at least some applications could not be safely re-executed with GO.) Also, it seems that by using TEX you are committed to it as your command-line interface, giving up the opportunity of using command-line history shells (LSH, EASE, BGHIST).

More generally, the TEX shell, and therefore any scheduled commands, will only run when TEX is the active shell. So, for example, while ZFILER is in use, command scheduling is suspended.

There is always some cost in performance in using a shell to process every command line. It's not very noticeable on a fast ram disk, but may be aggravating on a floppy system. This price is worth paying when the shell adds capability to most command-line entry operations (as command history or full-screen menu shells do). It proves more of a burden than a boon when the added features only occasionally get used (i.e., when a scheduled event's time arrives).

AT's rather different design, built around an RSX, enables it to watch the clock more or less continually, regardless of what is running. It accomplishes this without the overhead of reloading a shell for each command, but at the price of reduced user memory.

Shells also raise complications for AT, namely—how or whether AT can interrupt a shell to run a scheduled command. This question has just begun to be discussed among the alpha testers, but one promising idea (suggested by Jay Sage) is to terminate an active shell if it has had no console activity for several minutes. We'll be experimenting with several approaches. Stay tuned!

This short tour of recent CP/M system developments demonstrates a wealth of vigorous innovation. Perhaps in MS-DOS-land the true hacker is a dying breed, but in CP/M there is much left to hack. Tim Parker and others longing for the old challenge are most welcome to (re)join us and push the envelope further! •

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**

Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**

Auto-install Z-System (ZCPR v 3.4). Dynamically change memory use. Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

JetLDR **$20**

Z-System segment loader for ZRL and absolute files. (included with Z3PLUS and NZ-COM)

ZSDOS ..... **$75, for ZRDOS users just $60**

Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**

Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**

Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

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

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **JetLDR** ©, **JetFind** © Copyright 1986-88 by Bridger Mitchell.

The Z-System Corner

by Jay Sage

By the time you read this, summer vacation will probably be just a fond memory for you, but it is August as I write this, and I have just returned from three weeks in Israel. This was a total vacation. I didn't touch or even think about computers the whole time I was away, except at the very end when my mind started to refocus on the responsibilities that awaited me at home, including this TCJ column. It was so nice to get "computer compulsion" out of my system that I have not been all that eager to get back immediately to my old routine...but I know it will happen soon enough.

Having not thought about computing for a month, I had to work to recall the things I was excited about before I left and planned to discuss in this issue. Fortunately, I left myself some notes. One item was the continuation of the BYE discussion, this time covering the extended DOS functions implemented in BYE. The truth is I have neither the energy nor the time to take up that subject now. Instead, I am going to come back once again to my favorite subject: aliases and ARUNZ.

I think ARUNZ is the one thing that keeps me hooked on Z-System and not eager to follow after the MS-DOS crowd. Although I have looked hard, I have not found anything with the combined simplicity and power of ARUNZ for MS-DOS. The mainframe batch processing language REXX, which has been ported by Mansfield Software to DOS machines, is far more powerful, and some day I hope to port a greatly simplified version to Z-System. The DOS version of REXX, you see, takes over 200K of memory while it is running! That often does not leave enough memory, even on a 640K machine, for application programs to run. I think I actually have more problems running out of TPA on my Compaq 386 than I do on my SB180!

Anyway, for this column I am going to begin with a very brief report on a rather dramatic change in the works for ARUNZ and the Z-System as a whole. Then I am going to describe two ARUNZ applications that I recently developed for my own use. I think they illustrate some interesting general principles, and you may even find them useful as they are.

The Extended Multiple Command Line

Most people who use ARUNZ aliases—or even standard aliases—sooner or later run into a situation where the command line overflows and the whole process comes to a crashing halt (well, not really a crash, but a sudden stoppage). The standard Z-System configuration supports a multiple command line buffer (MCL) that can accommodate 203 characters. The largest size possible is 255 characters. Either way, there comes a time when aliases are invoked from command lines that already contain additional commands, and the combined command line is too long to fit in the MCL. People like Rick Charnes would have this happen constantly if they did not adopt a strategy to avoid the problem (more about that in one of our examples later).

I have long been intrigued by the possibility of having a much longer command line. The command processor (CPR) has always used a word-size (16-bit) pointer into the command line, and so, without any change in the ZCPR34 code, the CPR could handle a command line as big as the address space of the Z80.

To verify this, I performed a simple experiment. I configured a Z-System with free memory after the MCL, and then, using the memory utility program MU3, I manually filled in the command line with a very long multiple command line sequence terminated, as required, by a null character (binary zero). Sure enough, after exiting from MU3, the huge command line ran without a hitch.

The next step was to write a special new version of ARUNZ that could be configured to recognize an oversized MCL. Richard Conn set up the environment descriptor (ENV) with a one-byte value for the length of the command line that the MCL buffer could contain. Thus there is presently no way to support an extended MCL (XMCL) in a system-invariant way, that is, in a way that allows programs to determine at run time how big the MCL is. We are working on that problem right now, and, by the time you are reading this, there will almost certainly be a new ENV type defined (81H) that uses one of the remaining spare bytes in the type-80H ENV to report the size of XMCL to programs smart enough to check for it.

The original, single-byte MCL size value in the ENV has to remain as is and contain a value no larger (by definition) than 255 (0FFH). That value is used by the command processor when new user input is being requested. There is no way for the CPR to allow users to type in command lines longer than 255 characters without adding a vast amount of code to perform the line-input function now so conveniently and efficiently provided by the DOS. A shell could be written that included such code, but I really can't imagine anyone typing in such long command lines. If they do, it probably shows that they are not making proper use of aliases.

I have decided to use only one of the spare ENV bytes for the XMCL size and to let that value represent the size—in paragraphs—of the total MCL memory buffer allocated, including the five bytes used by the address pointer, size bytes, and terminating null. The term 'paragraph' as a unit of memory is not often used in the Z80 world. I believe it was introduced with the 8086 processor, where the segment registers represent addresses that are shifted four bits right. Each unit in the segment register is, therefore, 16 bytes and is called a paragraph. With this system, the XMCL buffer can be as large as 255 * 16 = 4080, which allows command lines with up to 4075 characters. Rich Charnes, do you think you can live with that without cramping your style too much?!

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER file maintenance shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (on PC-Pursuit). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Finally, Jay recently became the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail.

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog *computation to solve problems in signal, image, and information processing.*

Most people will not want to allocate that much memory to the operating system, and I would never have considered this step before the new dynamic versions of Z-System were available. While I might be willing to allocate 1K to the XMCL most of the time, I certainly would want to be able to reclaim that memory when I need it. I'm not sure whether NZCOM or Z3PLUS can be cajoled into handling this kind of flexibility yet; new versions may be needed at some time in the future.

I put the new version of ARUNZ out for beta test, and it worked just fine, and one could write very long alias scripts. Rick Charnes, however, quickly identified a problem. Suppose a conventional alias appeared in the command sequence. After expanding itself and constructing the new command line, the alias would find that, as far as it knew, there was not enough room for it in the MCL. In a nutshell, the hard part with going to the XMCL is that it is not enough to have an advanced ARUNZ; all programs that operate on the MCL must be upgraded. We hope to have new versions of the library routines in Z3LIB that perform these functions. Then, if we are lucky, most of the utility programs can be upgraded simply by relinking. I'm sure it won't be quite that easy, of course!

A MEX Alias

For those who are not familiar with it, MEX (Modem EXecutive) is an advanced telecommunications program written by Ron Fowler of NightOwl Software. Early versions were released for free to the public (up to version 1.14), while the most advanced versions (called MEX-Plus) are commercial products. I use version 1.65, and some of the specifics in my example apply to that version. I am pretty sure that the technique I describe can be applied to the free version as well.

Rather than being a telecommunications program, MEX should probably be considered a telecommunications programming language. It supports a very wide range of internal commands for managing telecommunications tasks, and it even has a script language for automating complex sequences of operations.

The MEX command line allows multiple commands to be entered just as in Z-System, and a MEX command allows the user to define the command separator. Although I depend on aliases to generate complex Z-System commands and MEX script files to automate complex MEX command sequences, I still frequently make use of simple, manually entered multiple commands.

Being accustomed as I am to entering Z-System commands separated by semicolons, I naturally set up my version of MEX to use the semicolon as its separator, too. Now I can comfortably work in both environments. However, I also frequently like to invoke MEX with some initial commands, which MEX allows one to include in the command tail. Here's a simple example.

```
B11:TEMP>mex read mnp on
```

This command invokes MEX and tells it to run the script file MNP.MEX with the parameter "ON". This script causes a string to be sent to my modem which engages the MNP error correcting mode (yes, when I purchased my most recent modem -- replacing a USR Password -- I decided to spend the extra money for MNP, although at the time there weren't many systems that supported it; now I'm glad I did).

That command line works fine. But often I want to do more, and so I always wanted to enter something like:

```
B11:TEMP>mex read mnp on;call zitel
```

This would start out by doing what the first example did but would then continue by placing a call to the ZITEL BBS. [If you can keep a secret, I'll tell you that the ZITEL BBS is the MS-DOS system that I run for the ZI/TEL Group of the Boston Computer Society. ZI/TEL comes from the letters in Zilog and Intel, and it symbolizes the fact that we support the two main operating systems run on chips from those companies: CP/M and MS-DOS. The BBS machine, a Kaypro 286/16, is sitting in the other room (you don't think I'd allow it in the same room with the Z-Node, do you?), and it has an HST 9600 bps modem with MNP error correction. If you want to contact me there, by the way, the number is 617-965-7046.]

An on-the-ball reader already realized that the above command will not work, because the semicolon separator before the CALL command, which I intended as the MEX separator, will be interpreted by the CPR as its separator, and it will terminate the MEX command. What can we do about this?

Some compromise here is inescapable, and I was willing to accept—from the CPR command line only—a MEX separator other than semicolon. Thus the following form would be acceptable

```
B11:TEMP>mex read mnp on!call zitel
```

with an exclamation point as the separator as in CP/M-Plus. But for years I could not figure out how to accomplish this.

At first I thought there was a very simple solution. When MEX starts up, it can be set up to automatically run an initialization script file INI.MEX. So, I created a version of MEX (the MEX "CLONE" command makes it easy to create new versions) that used "!" as the separator, and I created an INI.MEX file with the command

```
STAT SEP ";"
```

Thus, as soon as MEX was running, the separator would be set back to a semicolon. Unfortunately, to my chagrin, I learned that MEX invokes the INI.MEX script only when no commands are included on the command line. With the ZITEL command line shown earlier, MEX would be left with the exclamation point as the separator.

Here is what I thought of next (at least momentarily). Rename MEX.COM to MEX!.COM and set up a MEX alias in ALIAS.CMD with the definition

```
MEX   mex:mex! $*!stat sep ";"
```

The idea here is that the user's MEX commands from the command line (separated by "!") will be passed in by the $* parameter and will have the STAT command added. Thus our earlier example will turn into the command line

```
B11:TEMP>mex:mex! read mnp on!call zitel!stat sep ";"
```

This, of course, fails for the same reason that I could not just enter commands with semicolons in the first place. The trick to get around this is to use a command that for some reason Ron Fowler does not document in the MEX manual: POKE. It works like the Z-System command of the same name and places a byte of data into a specified memory address.

I knew the value I wanted to poke: 3BH, the hex value for the semicolon character. The question was, where should it go? To find out, I took a version of MEX set up with semicolon as the separator and the version with exclamation point as the separator and ran the utility DIFF on them (in verbose mode to show all the differences). Then I looked for the place where the former has a semicolon and the latter an exclamation point. For MEX-Plus this turned out to be 0D18H so that the MEX poke command would be

```
POKE $0D18 $3B
```

Note that MEX uses a dollar sign to designate hex numbers. The alias now read

```
MEX   mex:mex! $*!poke $$0d18 $$3b
```

Observe that a double dollar sign is needed to get a single dollar sign character into a command. A lot of people forget this and end up with scripts that don't do what they're supposed to.

I tested this, and it works splendidly—but with one possible

'gotcha'. The commands passed to MEX must not invoke any script files that depend on the command separator being a semicolon (because it will be exclamation point until the final poke command runs); nor may the read files change the command separator (because the rest of the command sequence still assumes it is the exclamation point). For this reason, it is prudent to write all script files with only one command per line so that no separator is needed. I haven't been doing this, but I will from now on!

One final word on the script. I actually did not do this exactly as I have described. Instead, I left my MEX.COM set up with the semicolon separator, and I created a distinct ARUNZ alias called MEX! so that I would be reminded of the separator. This alias script reads

```
MEX!  get 100 mex:mex.com;poke d18 "!;
          go $*!poke $$0d18 $$3b
```

This uses the famous poke&go technique originated by Bruce Morgen. MEX.COM is loaded into memory by the GET command, and then the Z-System POKE command sets "!" as the command separator. Then the modified loaded code is run by the GO command. The rest is as described previously.

A Spell-Check Alias

I try to remember to put all my writing through The Word Plus spelling checker that came with WordStar Release 4 so that as many typos as possible will be caught. The procedure for doing that on a Z-System is a bit complicated because the text file is generally not in the same user area as the spelling check program. While writing my last TCJ column, I finally got fed up with the complexity and automated the whole process using a set of aliases.

I wanted to support the following syntax:

```
C1:TCJ>spell filename.typ dictname
```

If just the file name was given, the alias would prompt for the name of the special dictionary to use, and if not even a file name was given, then the alias would prompt for both names. A special version of the command, ZSPELL, would take only the file name and would automatically use ZSYSTEM as the name of the special dictionary (it knows about mnemonics like ZCPR, MCL, and RCP, and about all those special words like debugger, relocatable, and modem). We'll describe the general alias set first. In listing the aliases, we will write them in multiline format for easy reading;in the ALIAS.CMD file the scripts have to be on a single line (though I hope that will change soon).

The user-interface alias, SPELL, deals only with the matter of how many parameters the user has provided. It reads as follows:

```
SPELL
   if nu $1;
      /TW0;
   else;
      if nu $2;
         /TW1 $1;
      else;
         /TW2 $1 $2;
      fi;
   fi
```

If no parameters at all are provided (IF NULL $1), then the secondary script TW0 is run. The leading slash signals ZCPR34 that the command should be directed immediately to the extended command processor. If a first parameter but no second parameter is present (IF NULL $2), then the secondary script TW1 is run. Finally, if both parameter are provided, then script TW2 is run.

The script TW1 includes a prompt only for the name of the special dictionary file:

```
TW1
   $"Name of special dictionary: "
   /TW2 $1 $'e1
```

The first token in any user response to the first prompt ($'E1 -- when working with ARUNZ you should have a printout of the parameter DOC file) is used along with the file name that was already given, and both are passed to TW2.

The script TW0 includes prompts for both the file name and the special dictionary:

```
TW0
   $"Name of file to check: "
   $"Name of special dictionary: "
   if ~nu $'e1
      /TW2 $'e1 $'e2
   fi
```

The first tokens in the responses to the prompts are passed to script TW2. If no file is specified for checking, the alias simply terminates.

Before we look at TW2, which does the real work, let me ask a rhetorical question: why do we break this process up into so many separate aliases. There are two main reasons. The first is that the command line buffer would overflow if all these smaller scripts were merged into a single big script. The extended MCL we discussed earlier could overcome this problem, but for another reason we would still have to use separate aliases.

As I have discussed in past columns, ARUNZ cannot know at the time it expands a script what the results of conditional tests will be later when the IF commands are run. Thus ARUNZ must process all user input prompts that appear in the script. This would mean asking for a file name and special dictionary even when the names were given on the command line. The solution to this problem is to put the prompts in separate scripts that get invoked only when the information requested in those prompts is actually needed.

Now let's look at the script TW2.

```
TW2
   path /d=tw:;
   $td1$tu1:;
   tw:tw $tf1 $tn2.cmp;
   path /d=;
   /twend $tn2;
   $hb:
```

This is simpler than what you expected, no? Well, there is still a lot of work imbedded in the subroutine script TWEND, which we will cover later. Here we broke up the script solely to prevent MCL overflow.

The first command makes use of the ZSDOS file search path (see the articles by Hal Bower and Cam Cotrill on ZSDOS in TCJ issues 37 and 38). Although there was an attempt to update WordStar Release 4 to include some Z-System support, no such attempt was made with The Word Plus spell checker. In general, the file to be spell-checked will be in one directory and The Word files in another directory. The main program TW.COM could be located by the command processor using its search path, but TW.COM needs a number of auxiliary files, such as the dictionary files. How can the system be made to find all of these files at the same time. ZSDOS provides the answer.

I have replaced the standard Z-System PATH command with the ZSDOS utility ZPATH (renamed to PATH). The first command in TW2 defines the DOS search path to include the directory TW:, which is where I keep all the files that are part of The Word Plus spell-checking package. Once that directory is on the DOS path, all files in it will more-or-less appear to be in the current directory. Very handy! If you use ZDDOS, the search path is not available. I will not show it here, but you can accomplish the same thing using only public files. It's just not quite as neat and straightforward. I am willing to pay the small memory penalty to get the nice extra features of ZSDOS over ZDDOS.

The second command logs us into the directory where the file to be checked resides. If we did not include a DIR: prefix, we were already there, but the extra command does not hurt, and it is nice to know that a directory can be specified explicitly (in either DU: or DIR: form) for the file to be checked. There could be a problem if the file is in a user area above 15, since you may not be able to log

into that area. My configuration of Z34 allows this, but when I run BGii I lose this feature (and I sure miss it). If you can't log into those areas, then you should not keep files there that you want to spell-check.

The third line actually runs the spell checker (you knew that had to happen some time!). Notice that even if the user specified a file type for the special dictionary, type CMP is used. Only the name ($TN2) without the type is taken from the user. As the master program TW.COM is run, it will find its component program files (e.g., SPELL.COM, LOOKUP.COM, MARKFIX.COM) and the various dictionaries in the TW: directory thanks to ZSDOS, and it will find the text file in the current directory. As it works through the text, if there are any questionable words, it will write out a file ERRWORDS.TXT to the current directory. If any words are added to the special or UPDATE dictionaries, then the modified dictionaries will be read from TW: but written out to the current directory. You must understand these facts in order to understand the rest of the script.

Once the spell-checking is complete, the ZSDOS path is set back to null (unless I have a special need to have the DOS perform a search, I leave it this way to avoid surprises). Then the ending script TWEND is run, and finally the original directory ($HB:) is restored as the current directory.

Now let's look at TWEND. As it is invoked, the name of the special dictionary is passed to it. TWEND's job is to clean up scratch files and to take care of any updated dictionaries. It reads

```
TWEND
   if ex errwords.txt;
      era errwords.txt;
   fi;
   /dupd $1;
   /dupd updict
```

For efficiency and to prevent MCL overflow, the dictionary updating is performed by yet another subroutine script, DUPD. It gets called twice, once with the special dictionary (if any) and once with the update dictionary. It reads as follows:

```
DUPD
   if ex $tn1.cmp;
      mcopy tw:=$tn1.cmp /ex;
   fi
```

If an updated version of the specified dictionary exists in the current directory, then it is copied to the TW: directory, overwriting any existing file of that name (MCOPY option E). The source file is then erased (MCOPY option X). Oh yes, I almost forgot; the MCOPY here is my renamed version of the COPY program supplied with ZSDOS.

That is it except for showing you the special ZSPELL version of the alias. Notice that I make the "ELL" part of the command optional by inserting the comma in front of that part of the alias name. I also allow the script to be invoked under the name ZTW. The main SPELL script actually has the name "TW=SP,ELL" on my system. Since TW: is not on my command search path, the command "TW" will invoke the ARUNZ script unless I am in the TW: directory at the time.

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- New Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($69.95)
  - NZ-COM: Z-System for CP/M-2.2 computers ($69.95)
  - ZCPR34 Source Code: if you need to customize ($49.95)
- Plu*Perfect Systems
  - Backgrounder II: switch between two or three running tasks under CP/M-2.2 ($75)
  - ZDOS: state-of-the-art DOS with date stamping and much more ($75, $60 for ZRDOS owners)
  - DosDisk: Use DOS-format disks in CP/M machines, supports subdirectories, maintains date stamps ($30 - $45 depending on version)
- BDS C — Special Z-System Version ($90)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Assembler Mnemonics: Zilog (Z80ASM, Z80ASM+), Hitachi (SLR180, SLR180+), Intel (SLRMAC, SLRMAC+)
  - Linkers: SLRNK, SLRNK+
  - TPA-Based: $49.95; Virtual-Memory: $195.00
- NightOwl Software MEX-Plus ($60)

Same-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $4 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (password = DDT)(MABOS on PC-Pursuit)


```
ZTW=ZSP,ELL
   if nu $1;
      /ZTW1;
   else;
      /TW2 $1 zsystem.cmp;
   fi

ZTW1
   $"Name fo file to check: "
   if ~nu $'el
      /TW2 $'el zsystem.cmp
   fi
```

I hope you find these alias examples useful and instructive. That's all for this time. See you again in two months. •

Programming Input/Output With C

Part 2: Disk and Printer Functions

by Clem Pepper

The "stream" concept and the definition of FILE as a structure defined in <stdio.h> were described in part 1. The application there is with respect to the keyboard, stdin, and the screen, stdout. In this writing a stream defines the connection to a file (disk file or a device such as a printer) which makes the file appear to be a sequence of bytes.

The multiplicity of file input/output related library functions can appear overwhelming. This, in part, because I/O files exist at two levels--lower and upper. The lower level functions are unbuffered and operate with data bytes. The upper level are buffered and operate with blocks of data. In addition, under MS DOS, there are differing standards for two types of files: text and binary. (Under UNIX a single standard applies to both types.)

The typical sequence in reading a disk text file is to (1) open the file, (2) read the content, and (3) close the file. We typically will do more than that, depending on how we plan to use the file content.

Files that we wish to work with may be called from the command line or from within our program. Each has its place. We will begin with an overview of command line procedures. We will then look briefly at low level I/O functions before delving rather deeply into the higher level, which are those we commonly work with.

Command Line Arguments

There are two approaches to reading in disk files from the command line. The most general, offering the best flexibility, is the use of "argc" and "argv" as arguments to main(). However we can also read a disk file by redirection.

My introduction to C several years back was with The Software Toolworks C/80. For a long time I was baffled by the meaning of

```
main(argc,argv)  /* command line arguments */
int argc;        /* number of arguments    */
char *argv[];    /* argument name strings  */
{
```

"argc" stands for "argument count." It defines the number of arguments on the command line. The "command line", by the way, is simply the instruction we enter on the keyboard. When we wish to clear the screen, we type CLS and press the ENTER key. CLS is the command line. When we use our word processor we enter the command line followed with a single argument; the first, the command line, calls the editor and a second entry, the command line argument, names the file.

"argv" stands for "argument values." (I am sure I could have come up with something a lot better here.) "argv" represents an array of strings, one argument per string. The first string is 'argv[0]' as the first element of an array begins with the '0' subscript. Right here is an excellent source of confusion, as argc has a value of 2 when two argument values, 0 and 1, are present.

We are not compelled to use argc and argv, and, as a matter of fact, I do not. I use

```
main(num,fname)
int num;          /* in place of argc  */
char *fname[];    /* in place of argv  */
{
```

A formal argument declaration employing the array symbol [] as in "char *fname[]" is the same as assigning a pointer; that is, we could also write "char **fname." Either form declares fname (argv) to be a pointer to a pointer to a char.

Listing 1 is a program illustrating command line redirection (ASC_VAL < filename). This program reads a text file and displays each character with its ASCII value. Listing 2 illustrates main(argc,argv) for command line input. This program reads a text file from disk and writes it to an array. The array content is then written to a designated disk file while also displaying its content on the screen. This program thus requires two files as its argument: the source and a destination. These may be from or to another directory.

```
/*=====================================================**
** ASC_VAL.C  Adapted from Ref 1, p 60                  **
** Reads a file and displays its ASCII char values.     **
** Uses command line redirection: ASC_VAL < filename    **
** Compiled using TURBO C Ver. 1.5                      **
=====================================================*/

#include <stdio.h>
#define CNZ '\032'      /* Ctrl Z */

main()
{
  int ch;
    printf("Text Mode:\n");
     do  {
        ch = getchar();
         switch(ch)  {
            case '\n' : printf("LF : ");  break;
            case '\r' : printf("CR : ");  break;
            case CNZ  : printf("^Z : ");  break;
            case EOF  : printf("EOF : ");  break;
            default   : printf("%c : ",ch);
                      }
           printf("%3d\n",ch);
         } while (ch != EOF);
      fclose("stdin");
     exit(0);
}
```

Listing 1. A program to read a disk file and display the ASCII value of its characters.

Low Level File Functions

The descriptions found in the majority of C programming texts hardly touch on the subject of the low level functions, if they do at all. This because the higher level functions are the easiest and fastest to use with the least possibility of programming errors. In fact, among the books in my C library, only one, "Advanced C Primer ++"[1] provides a detailed description of the lower level functions.

When we look at the source code for the higher level functions we discover they rely heavily on the lower level functions. This is not too surprising.

Table 1 summarizes the lower level functions. Definitions and program examples in this writing are based for the most part on Borland's TURBO C, version 1.5. I purchased the library source

```
/*==================================================**
** COUNT1.C   From ref 1 p 66, Advanced C.          **
** Counts chars in a file, one at a time.           **
** Compiled using TURBO C Ver. 1.5                  **
**==================================================*/

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];   /* Use command line to get filename */
{
  char ch;      /* Place to hold each char as read  */
  int fd;       /* file descriptor - identifies file */
  long count = 0;

  if(argc != 2)
   { printf("Requires a command line argument."); exit(1);  }

    fd = open(argv[1],0);
      while(read(fd, &ch, 1) > 0)  count ++;
        close(fd);

    printf("File %s has %ld characters.\n", argv[1],count);
     exit(0);
}
```

Listing 2. A program for counting characters in a file and writing the count to the screen.

```
/*==============================================**
** FI_ARRY.C by Clem Pepper                     **
** Entering disk file text into an array for    **
** writing to the screen and a new file.        **
** Compiled using TURBO C Ver. 1.5              **
**==============================================*/
#include <stdio.h>
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
#define CMD_LN_ERR  "Source and destination files required.\n"

/* == Begin program == */
main(num,fname)
int num;
char *fname[];
{
 FILE *in, *out;   /* input and output streams */
  char file_in[880];     /* maximum array size */
  int ch, i = 0, n = -1;
   puts(CLRSCRN);          /* clear the screen */

if(num != 3) { puts(CMD_LN_ERR);
  exit(0);    }

/* == open the source file for reading == */
 else if((in = fopen(fname[1],"rt")) != NULL)
                        {
                      {
/* == read the source file into array file_in[] == */
   while((ch = fgetc(in)) != EOF)    /* get char from in */
                    {
     file_in[i++] = ch;
                    }
                      }
/* == close the source file == */
          fclose(in);
                        }
/* == Convert final char to '\0'; transfer count  == */
     file_in[i] = '\0';

/* == assign the destination file for writing == */
    out = fopen(fname[2],"wt");

/* == write to the screen and the destination file == */
 while(n++ != i && file_in[n] != '\0')   {
   putchar(file_in[n]);
    fputc(file_in[n],out);                }

/* == close the destination file == */
     fclose(out);
       exit(0);
}
```

Listing 3. A program which reads a text file into an array and writes it to the screen and a disk file.

codes for this, and also for Mix's POWER C. Both compilers conform closely to the projected ANSI standard. That is, I have compiled a number of programs on both with only minor changes required. Even so, their library files read significantly different. The low level library functions for TURBO C, for instance, are written with large amounts of inline assembly, which is not the case for POWER C.

We must open a disk file before we can perform work on it. When our work is complete the file must be closed. The two lower level functions for these tasks are open() and close(). Listing 3, COUNT1.C, illustrates open(), read(), and close(). In this program the type long variable count is counting the elements of array argv[]. Note that a FILE declaration is not required.

Sometimes there is a need to save the file for further activity. That need did not exist in the example just used. Suppose we look at a situation where we want to save the file so as to make use of it later in the program. The following lines read the source code character at a time and save them in the array file_in[].

```
while((ch = getc(in)) != EOF)     /* get char from in */
            {
   file_in[i++] = ch;
            }
               }
      fclose(in);
                  }
/* ** Replace final char with '\0'; transfer count  ** */
   file_in[i] = '\0';
```

The manner in which getc() reads the input stream character by character should be understood. As each characters is read it is written into the array file_in[i]. The array variable "i" is incremented following each array entry. (C's ability to combine operations in this way can be confusing.) The file is read until the EOF is detected, at which point the file is closed. One operation remains: to replace the EOF with the required '\0'.

By saving the file in an array we are able to access any specific character through its element number. In Listing 2 we simply write the array content out to the screen and to a designated disk file. However there are often situations in which additional operations are needed. This example, as an illustration, is taken from a typing tutor program I am working on. There the array content is read back to the upper part of the screen. At the end of each line, detected by testing for '\r', a tilde (~) is added to inform the user that a carriage return should be made. The leading character is then intensified to mark the beginning of the text and an instruction displayed for the student to begin typing the file. We can see where it would be quite difficult to conduct all these operations without the benefits of array data storage.

Listing 4, REVERSE.C, employs open(), lseek(), read(), and write(), to read a file and print it out in reverse order. Again, note that a FILE declaration is not required.

High Level File Functions

There is very little reason for us to write application programs with the low level functions. There is a great deal of reason not too. Which of course explains why application programs are written using the high level functions. Table 2 is a summary of the major high level functions. Notice the similarity of many of these to the keyboard functions defined in part 1 of this series. In fact we can often elect to use getc() or gets() in lieu of fgetc() or fgets() in our programs.

Basically there are three operations we might wish to perform on a file after opening it. These are 1) read the file, 2) write to a new file, or 3) append to an existing file. The initial step, of course, is to open the file. The function call for fopen() requires two arguments: the first to name the file, the second to define the mode in which the file is to be opened.

The file to be opened may be identified by redirection, with a command line entry or within the program. If our program has

```
/*=================================================================**
** REVERSE.C    Ref: Advanced C p73                                **
** Uses write() and lseek() to print out a file in reverse order.  **
** Compiled using TURBO C Ver. 1.5                                 **
**=================================================================*/

#include <stdio.h>
#include <fcntl.h>       /* used by open() */
#define CNTL_Z '\032'    /* text mode EOF */

main(argc,argv)
int argc;
char *argv[];
{
  char ch;  int fd;
  long count, last, lseek();

  if(argc != 2)  {
    printf("Useage:reverse filename\n");  exit(1);
                 }
  if((fd = open(argv[1],O_RDONLY | O_BINARY)) < 0)
                    /* read only and binary modes */
                 {
    printf("reverse can't open %s\n", argv[1]);  exit(1);
                 }
     last = lseek(fd,0L,2);   /* go to end of file */

     for(count = 1L; count <= last; count ++)
      {
       lseek(fd,-count,2);  /* go backwards */
       read(fd,&ch,1);
       if(ch != CNTL_Z && ch != 'r')
          write(1,&ch,1);  /* 1 is the screen */
       }
   close(fd);
  exit(0);
}
```

Listing 4. A program read a short text and write it back to the screen in reverse order.

```
/*================================================
** LINE_PRN.C  by Clem Pepper                     **
** A program for inserting line numbers into     **
** a C file before displaying or printing it.     **
** Compiled using TURBO C Ver. 1.5                **
================================================*/
#include <stdio.h>
#include <dos.h>

#define MAXLEN 81

/* ** ==        Begin program       == ** */
main(num,fname)  /* num == argc,  fname == argv */
int num;         /* Looks for two files on command line */
char *fname[];   /* file name to be printed */
{
 FILE *in;       /* Pointer to file input stream */
  int line_cnt = 0;     /* line counter */
  char *string[MAXLEN]; /* array holding program line */

/* ** Test for filename; print message and exit if missing ** */
   if(num != 2) { printf("A source file is required.\n"); exit(0); }

/* ** Print file name information line; open file ** */
    printf("\nFile being printed is %s\n\n",fname[1]);
    in = fopen(fname[1],"rt");          /* Open file for reading */

/* ** increment line counter, read next line, display & print ** */
     while(fgets(string,MAXLEN,in) != NULL)
                    {
     line_cnt += 1;
      printf("%d %s",line_cnt,string);
                     }
       fclose(in);  /* Close the disk file */
         exit(0);
}
```

Listing 5. A program for inserting line numbers into a text file.

application to many files we would prefer to use the command line. If the program is unique to a specific task we most likely will identify it within the source code.

A command example is:

```
main(argc,argv)  /* command line arguments */
int argc;        /* number of arguments    */
char *argv[];    /* argument name strings  */
   {
   FILE *in;
   in = fopen(argv[1],''r'');
```

where "in" is the stream pointed to and "r" is the mode in which the file is opened.
We may specify one of three basic modes for fopen(): r, w, and a.

- r Opened for reading only. The file must exist or a NULL is returned.
- w Opened for writing. The file need not exist. If it does exist the CONTENT IS DELETED, so be careful.
- a Opened for writing. If the file exists new text is added at the end. If the file does not exist it is treated as though opened in the "w" mode.

There are options to these. Appending a "t", i.e., "rt", specifies the file is to be opened in the text mode. Similarly, appending a "b" indicates the binary mode. If neither is appended, the mode is governed by global variables defined by O_... constants in the library file fcntl.h.

Appending a + (r+, w+, a+, rb+, etc.) opens the file for read/write updating. The "TURBO C Reference Guide" [2] states that when a file is opened for updating both input and output may be requested. However there is a requirement for intervening calls to flush the buffer and reset the file pointer when changing from the one to the other.

In summary, fopen() opens the specified file, creates a buffer to hold blocks of data from the file and creates the structure of type FILE describing the file and the buffer. The return value is the pointer "*in" to the FILE structure. In other words, the data stream designated "*in" is read into a buffer defined by the structure of type FILE. Future reference to the file is by "*in", not the name employed in argv[].

Before attempting to read from or write to the file we will want to test for the correct command line entries. The minimum number of arguments for a disk file are 2: our program name followed by the disk file name. If the program name alone is entered the user must be alerted and the program aborted. An example of a condition test when in the read mode is:

```
if(name == NULL) { puts(''A source file is required.'');
   exit(0); }
In the write mode the test might be:
if(name == NULL) { puts(''Cannot open destination file.'');
   exit(0); }
```

What we do next depends on the application. In the simplest case we may simply wish to display the text on our screen as it is read in or perhaps pass it on to the printer. On occasion we will create a new file or modify an existing one.

Suppose as an opening illustration we read in a short file, insert a line number at the beginning of each new line and pass it on to the screen. Listing 5, LINE_PRN is both an illustrative example and a useful utility. By the use of command line redirection (or CTRL-P for printing) the text with the line number inserted can be written to a new file or printed out.

For line number insertion we need two additional dec-

```
/*************************************************************
** PRN_SET.C  by Clem Pepper                                **
** A program for printer code selection applicable to       **
** Epson and other printers responding to the Epson codes.  **
** Compiled using TURBO C Ver. 1.5                          **
*************************************************************/

#include <stdio.h>
#include <dos.h>
#define VIDEO 0x10 /* interrupt 10H - screen functions      */
#define SFKEY 0x16 /* interrupt 16H - keyboard I/O          */
#define PRNT  0x17 /* interrupt 17H - parallel printer I/O  */

 /* === useful global declarations === */
static int col;       /* column variable   */
static int row;       /* row variable      */

     /* ***************************** */
     /* == Begin utility functions == */
     /* ***************************** */

/* == send desired codes to the printer == */
void prnt_code(char pchr)
{
  union REGS regs;   /* dos.h union          */
    regs.h.ah = 0;   /* print char in AL     */
    regs.h.al = pchr; /* char to be printed */
    regs.h.dh = 0;   /* port 0               */
    regs.h.dl = 0;
     int86(PRNT, &regs, &regs);
}

/* == screen clear == */
clr_scrn()
{
 union REGS regs;   /* dos.h union           */
  regs.h.ah = 6;    /* scroll up 25 lines    */
  regs.h.al = 0;    /*  when al is = 0.      */
  regs.h.ch = 0;    /* top row               */
  regs.h.cl = 0;    /* upper left screen col */
  regs.h.dh = row;  /* row to scroll up from */
  regs.h.dl = col;  /* col to scroll from    */
  regs.h.bh = 7;    /* attribute byte        */
   int86(VIDEO, &regs, &regs);
}

/* == position cursor at row and col values == */
void pos_cur(col,row)
{
  union REGS regs; /* dos.h union         */
    regs.h.ah = 2; /* set cursor position */
    regs.h.dh = row;
    regs.h.dl = col;
    regs.h.bh = 0; /* video page 0 */
     int86(VIDEO, &regs, &regs);
}

/* == Begin program == */
main()
{
 char pchr;
 col = 79; row = 24; pos_cur(col,row);
  clr_scrn();
 col = 0; row = 0; pos_cur(col,row);
  menu();
   exit(0);
}
/* == The leading graphic is made from ALT 199 == */
#define BONDRY "L-------------------------------------------------"

/* == display printer options menu == */
menu()  /* From main() */
{
 col = 10; row = 0; pos_cur(col,row);
   printf("PRESS FUNCTION KEY FOR DESIRED PRINTER OPTION");
 row += 1; pos_cur(col,row);
   printf(BONDRY);
 row += 1; pos_cur(col,row);
/* == The leading graphic is made from ALT 199 == */
   printf("L F1: Near Letter Quality ON");
 row += 1; pos_cur(col,row);
   printf("L F2: Near Letter Quality OFF");
 row += 1; pos_cur(col,row);
   printf("L F3: Elite Print ON");
 row += 1; pos_cur(col,row);
   printf("L F4: Elite Print OFF");
 row += 1; pos_cur(col,row);
   printf("L F5: Compressed Print ON");
 row += 1; pos_cur(col,row);
   printf("L F6: Compressed Print OFF");
 row += 1; pos_cur(col,row);
   printf("L F7: Compressed Elite Print ON");
 row += 1; pos_cur(col,row);
   printf("L F8: Compressed Elite Print OFF");
 row += 1; pos_cur(col,row);
   printf("L F9: Emphasized Print ON");
 row += 1; pos_cur(col,row);
   printf("L F10: Emphasized Print OFF");
 row += 1; pos_cur(col,row);
   printf(BONDRY);

    col = 12; row = 15; pos_cur(col,row);
       select();
}

#define MODE1A "Is Near Letter Quality ON correct? <Y/N>: "
#define MODE1B "Is Near Letter Quality OFF correct? <Y/N>: "
#define MODE2A "Is Elite Print ON correct? <Y/N>: "
#define MODE2B "Is Elite Print OFF correct? <Y/N>: "
#define MODE3A "Is Compressed Print ON correct? <Y/N>: "
#define MODE3B "Is Compressed Print OFF correct? <Y/N>: "
#define MODE4A "Is Compressed Elite Print ON correct? <Y/N>: "
#define MODE4B "Is Compressed Elite Print OFF correct? <Y/N>: "
#define MODE5A "Is Emphasized Print ON correct? <Y/N>: "
#define MODE5B "Is Emphasized Print OFF correct? <Y/N>: "

/* == make printer mode selection == */
select()  /* From menu() */
{ int i = 80; int choice_s, choice_a; int pflag = 0;
  char verify;
    choice_s = getch(); choice_a = getch();
      if(choice_s != 0) exit(1);
else if(choice_a == 59)  { printf(MODE1A); pflag = 1; }
else if(choice_a == 60)  { printf(MODE1B); pflag = 2; }
else if(choice_a == 61)  { printf(MODE2A); pflag = 3; }
else if(choice_a == 62)  { printf(MODE2B); pflag = 4; }
else if(choice_a == 63)  { printf(MODE3A); pflag = 5; }
else if(choice_a == 64)  { printf(MODE3B); pflag = 6; }
else if(choice_a == 65)  { printf(MODE4A); pflag = 7; }
else if(choice_a == 66)  { printf(MODE4B); pflag = 8; }
else if(choice_a == 67)  { printf(MODE5A); pflag = 9; }
else if(choice_a == 68)  { printf(MODE5B); pflag = 10; }

    verify = toupper(getch());

      if(verify == 'N')                              {
       col = 12; row = 15; pos_cur(col,row);
          while(i--)   { putchar(' '); col += 1; }
           col = 12; pos_cur(col,row); select();   }
             else if(verify == 'Y') set_mode(pflag);
               return;
}

/* == execute printer mode instruction == */
set_mode(int flg)  /* From select() */
{ char pchr;

/* == the modes require ESC as initial input == */
   pchr = '\033'; prnt_code(pchr);   /* escape     */

/* == turn on Near Letter Quality print == */
  if(flg == 1) {
  pchr = 0x78;   prnt_code(pchr);   /* char 'x'   */
  pchr = 0x1;    prnt_code(pchr);   /* decimal 1  */
   return;      }
/* == turn off Near Letter Quality print == */
  if(flg == 2) {
  pchr = 0x78;   prnt_code(pchr);   /* char 'x'   */
  pchr = 0x0;    prnt_code(pchr);   /* decimal 0  */
   return;      }
/* == turn on elite print == */
  if(flg == 3) {
  pchr = 0x4D;   prnt_code(pchr);   /* char 'M'   */
   return;      }
/* == turn off elite print == */
  if(flg == 4) {
  pchr = 0x50;   prnt_code(pchr);   /* char 'P'   */
   return;      }
/* == turn on compressed print == */
  if(flg == 5) {
  pchr = 0x0F;   prnt_code(pchr);   /* decimal 15 */
   return;      }
/* == turn off compressed print == */
  if(flg == 6) {
  pchr = 0x12;   prnt_code(pchr);   /* decimal 18 */
   return;    }
/* == turn on compressed elite print == */
  if(flg == 7) {
  pchr = 0x4D;   prnt_code(pchr);   /* char 'M'   */
  pchr = 0x0F;   prnt_code(pchr);   /* decimal 15 */
   return;      }
/* == turn off compressed elite print == */
  if(flg == 8) {
  pchr = 0x50;   prnt_code(pchr);   /* char 'P'   */
  pchr = 0x12;   prnt_code(pchr);   /* decimal 18 */
   return;      }
/* == turn on emphasized print == */
  if(flg == 9) {
  pchr = 0x45;   prnt_code(pchr);   /* char 'E'   */
   return;      }
/* == turn off emphasized print == */
  if(flg == 10) {
  pchr = 0x46;   prnt_code(pchr);   /* char 'F'   */
   return;      }
}
```

Listing 6. A program for writing control codes to a dot matrix printer.

Table 2: High level I/O function definitions

clearerr()
#include <stdio.h>
void clearerr(FILE *stream)
Resets the stream's error and end-of-file indicators.

fclose()
#include <stdio.h>
int fclose(FILE *stream)
Closes the named stream. All associated buffers are flushed prior to closing.

fcloseall()
#include <stdio.h>
int fclose(void)
Closes all open streams except stdin and stdout.

feof()
#include <stdio.h>
int feof(FILE *stream)
A macro that tests the given stream for an end-of-file indicator. Once the indicator is set read operations return the indicator until cleared by rewind() or the file is closed.

ferror()
#include <stdio.h>
int ferror(FILE *stream)
A macro that tests the given stream for a read or write error. Once set, the indicator remains set until cleared by clearerr() or rewind(). Returns non-zero if an error is detected on the named stream.

fflush()
#include <stdio.h>
int fflush(FILE *stream)
Flushes any output buffer associated with stream. Returns non-zero if an error occurs.

fgetc()
#include <stdio.h>
int fgetc(FILE *stream)
Returns the next character from the input stream. Performs the same as getc() except that getc() is a macro where fgetc() is a true function.

fgetchar()
#include <stdio.h>
int fgetchar(void)
A function the same as fgetc(stdin).

fgets()
#include <stdio.h>
char *fgets(char *string,int n,FILE *stream)
Reads at most n-1 characters from stream into string. Stops reading at n-1 or a newline char. The newline char is retained. The last character read into string is followed by a null char.

filelength()
long filelength(int handle)
Returns the length in bytes of the file associated with handle.

fileno()
#include <stdio.h>
int fileno(FILE *stream)
A macro which returns the integer file handle for a given stream.

findfirst()
#include <dir.h>
#include <dos.h>
int findfirst(char *pathname,struct ffblk *ffblk)
Makes a search of a disk directory by using the MS DOS system call 0x4E. Pathname is a string with an optional drive specifier, path and file name of a file to be found.

findnext()
#include <dir.h>
int findnext(struct ffblk *ffblk)
Is used to fetch subsequent files which match the pathname given in findfirst(). ffblk contains information needed to continue the search. findfirst() and findnext() each return 0 following a successful file match. When no more files or an error exists -1 is returned and errno() set to one of the following:

ENOENT Path or filename not found.
ENMFILE No more files.

fopen()
#include <stdio.h>
FILE *fopen(char *filename, char *type)
Opens the file defined by filename and associates a stream with it. Returns a pointer for stream identification in later operations. The type string is one of the following:

r open for reading only
w create for writing
a append: open for writing at end of file or to create a new file if none exists.

Attach a t for text, b for binary, i.e., rt, wb, etc.

fprintf()
#include <stdio.h>
int fprintf(FILE *stream, char *format[,argument,...])
Sends formatted output to a stream. Same features as printf().

fputc()
#include <stdio.h>
int fputc(int ch, FILE *stream)
Puts a character on a stream. Same features as putc().

fputs()
#include <stdio.h>
int fputs(char *string, FILE *stream)
Puts a string on a stream. Same features as puts().

fread()
#include <stdio.h>
int fread(void *ptr,int size,int nitems,FILE *stream)
Reads nitems of data of length size bytes from the defined stream into a buffer pointed to by ptr.

fscanf()
#include <stdio.h>
int fscanf(FILE *stream,char *format[,argument,...])
Provides formatted input from a stream. Same features as scanf().

fseek()
#include <stdio.h>
int fseek(FILE *stream,long offset,int fromwhere)
Sets the stream file pointer to a new position offset bytes beyond a location given by fromwhere. Clears the end-of-file indicator.

fromwhere	File Location
SEEK_SET (0)	file beginning
SEEK_CUR (1)	current file pointer position
SEEK_END (2)	end-of-file

fstat()
#include <stat.h>
int fstat(char *handle,struct stat *buff)
Obtains open file information.

ftell()
#include <stat.h>
int ftell(FILE *stream)
Returns the current file pointer.

fwrite()
#include <stdio.h>
int fwrite(void *ptr,int size,int nitems,FILE *stream)
Writes nitems of data of length size bytes to the defined stream into a buffer pointed to by ptr.

larations; one for an integer counter, initialized to zero, for numbering the lines, and another for declaring a character string of some maximum length. For screen display we do not want to exceed 80 characters. The two declarations then are:

```
#define MAXLEN 80
int line_cnt = 0;
char *string[MAXLEN];
```

The routine for reading a line, inserting its number, incrementing the line counter and displaying the line is:

```
in = fopen(fname[1],''rb'');
   while(fgets(string,MAXLEN,in) != NULL)
                {
      line_cnt += 1;
         printf(''%d %s'',line_cnt,string);
                }
            fclose(in);
```

The reference to NULL is with respect to the '\0' character termi-

nating the line. fgets() adds the '\0' when it detects a carriage return or on reaching the limit set by MAXLEN. While the loop is active fgets() reads a character from the stream "in" (file fname[1]) and saves it in memory at the location pointed to by "string." Note that one full line is being read. At the line's end the line counter is incremented. Then its value is printed followed by the string. fgets() detects the EOF character denoting end of the file and terminates the read. fclose() then closes the file and the program goes on to another activity, or exits.

Presetting Printer Modes from the Keyboard

When I acquired my first printer, an Epson MX-80 I soon wished for a simple way to pre-set its type mode from the keyboard. It could be done with BASIC but I sure didn't care for the hassle of loading and running BASIC just for that. I solved the problem then by writing four short routines in 8080 assembly to set and reset the compressed and emphasized modes, all then available for the MX-80.

The printers available today enable presetting a variety of modes directly at the printer by manipulating the printer's "ON-LINE", "FF", and "LF" keys. This can be aggravating, in particular if the printer is not close by our keyboard. Particularly so when we consider the printer mode will be changed by any program we happen to run that implements its own controls.

I find it super great to set a given mode whenever needed by a simple instruction from the keyboard. The program PRN_SET.C is given in Listing 6. PRN_SET permits setting or resetting any one of five modes from the ten function keys on our keyboard. Your modes may not agree with mine, but the principles described here should enable you to set your printer to any mode desired.

When reading from or writing to a disk file the FILE declaration provides the set ups for us through the library I/O functions. For writing to the printer we do not employ FILE. Instead DOS provides an interrupt just for parallel printers, INT 17H.

A necessary first step is to review our printer manual. Though I'll be referring to the Epson its codes are widely used. I have used PRN_SET with several Epson and Panasonic models. Of course a specific mode can be set only if the printer features it.

The manuals I have provide the mode codes in the appendix. Typical codes begin with ESCAPE (ESC), but not all. The emphasized mode, for instance, responds to ASCII 15 alone. Making a table such as that below is a good beginning.

```
MODE                  CPI        ON          OFF
----                  --         --          ---
Near Letter Quality    10     ESC ''x'' 1    ESC ''x'' 0
Elite                  12     ESC ''M''      ESC ''P''
Compressed             17         15          18
Compressed Elite       20     ESC ''M'' 15   ESC ''M'' 18
Emphasized             10     ESC ''E''      ESC ''F''
Note: CPI is ''Characters Per Inch.''
```

Although the compressed mode does not require a a preceding ESC it does not object to finding one, which simplifies the program.

Editor's note: The manual for my old Epson MX-80 shows the use of the escape code (1B hex) only with the printable ASCII characters. Control codes, such as the control O (0F hex, or 15 decimal) used for compressed do not require the escape delimiters because they are not printable characters.

Our next step is to become familiar with our operating system, in this instance MS DOS. Unfortunately, the standard MS DOS manuals provided do not tell us what we need to know. We either have to purchase the Programmer's Utility Pack or its equivalent, or a good text on MS DOS like Robert Jourdain's "Programmer's Problem Solver for the IBM PC, XT & AT."[3]

A parallel printer Input/Output interrupt, INT 17H, is provided for printer I/O. The interrupt employs two of the 8088 CPU registers, AX and DX, in their eight bit modes; that is: AH, AL, DH and DL. We must write a zero (0) to AH, then place the character to be printed in AL. MS DOS provides for up to three parallel printers with port assignments of 0, 1 and 2. Normally the assignment is to port 0, and we enter that into DH. We also enter a zero in DL. These are shown in the "void prnt_code(char pchr)" function following. I have compiled this with both Borland's Turbo C and Mix's Power C with minor differences so expect the same will be true for other compilers conforming to the proposed ANSI standards.

The library function int86 executes routines associated with the I/O interrupts. The function is:

```
int86(int interrupt, union REGS*inregs, union REGS*outregs);

The function I wrote for PRN_SET is

#include <dos.h>
#define PRNT 0x17  /* interrupt 17H - par. printer I/O */
/* == send desired codes to the printer == */
void prnt_code(char pchr)
{
   union REGS regs;    /* dos.h union          */
      regs.h.ah = 0;    /* print char in AL     */
      regs.h.al = pchr; /* char to be printed   */
      regs.h.dh = 0;    /* port 0               */
      regs.h.dl = 0;
         int86(PRNT, &regs, &regs);
}
```

Now that we know how to communicate with the printer we can go about configuring the communication between ourselves and the computer. This consists of 1) deciding on a screen menu format, and 2) the keyboard input for responding to the menu options. It is also advantageous to include an "oops" recovery from a wrong press on a function key.

A menu of printer options must be provided. These are printed using standard printf() statements. The menu text for Near Letter Quality is shown below.

```
printf(''PRESS FUNCTION KEY FOR DESIRED PRINTER OPTION'');
printf(''F1: Near Letter Quality ON'');
printf(''F2: Near Letter Quality OFF'');
```

The user is instructed to press one of the ten function keys to select the desired mode. Two calls to getch() are made: "choice_s" and "choice_a." This, if we recall from part 1, is because the non-ASCII keys have two responses, the first of which is zero. So if choice_s does not return a zero a wrong key has been pressed and the program exits. Choice_a corresponds to one of the ten keys. Keys F1 and F2 have codes 59 and 60, respectively. (Ref Table 1 of Part 1.)

Pressing a function key causes two things to happen: the printing of a query message on the screen, and the setting of a printer flag, pflag.

```
#define MODE1A ''Is Near Letter Quality ON correct? <Y/N>:''
#define MODE1B ''Is Near Letter Quality OFF correct? <Y/N>:''
```

Defines are often a convenient way to print strings. The query offers the user a means to correct a press of the wrong key.

```
/* == make printer mode selection == */
select()  /* From menu() */
{ int i = 80; int choice_s, choice_a; int pflag = 0;
   choice_s = getch(); choice_a = getch();
      if(choice_s != 0) exit(1);
else if(choice_a == 59)  { printf(MODE1A); pflag = 1; }
else if(choice_a == 60)  { printf(MODE1B); pflag = 2; }
```

Variable "pchr" carries the instruction to the printer. Two or three transmissions are required. The first, universal for all modes, is ESC, shown here as octal '\033'. Then, depending on the value of the flag variable one or two more assignments to pchr are transmitted via the union REGS regs;.

```
/* == execute printer mode instruction == */
set_mode(int flg)  /* From select() */
{ char pchr;

/* == the modes require ESC as initial input == */
   pchr = '\033'; prnt_code(pchr);   /* escape     */

/* == turn on Near Letter Quality print == */
   if(flg == 1) {
   pchr = 0x78;   prnt_code(pchr);   /* char 'x'   */
   pchr = 0x1;    prnt_code(pchr);   /* decimal 1  */
      return;      }
/* == turn off Near Letter Quality print == */
   if(flg == 2) {
   pchr = 0x78;   prnt_code(pchr);   /* char 'x'   */
   pchr = 0x0;    prnt_code(pchr);   /* decimal 0  */
      return;      }
```

Summary

This article has covered a considerable amount of material. We have learned that two levels of file input/output exist and how to program in either. Also that the command line arguments and redirection provide considerable flexibility in reading from and writing back to disk files. And then we saw that writing to the printer is simplified by the provision of interrupt 17H.

And that's it, mostly. Of course it doesn't just happen. A fair amount of time and effort must be invested as we descend into the depths of our machine's inner workings. The reward is a great deal of satisfaction in expanding our program activity beyond the keyboard and screen. ●

Reference Listing for PROGRAMMING INPUT/OUTPUT WITH C

1. "Advanced C Primer++" by Stephen Prata,
The Waite Group
Howard W. Sams & Co, Indianapolis, IN
2nd printing 1986

This text, chapter 3, "Binary and Text File I/O" provides a very thorough coverage of the low and high level file I/O functions.

2. "TURBO C Reference Guide"
Borland International, Inc.
4585 Scotts Valley Drive
Scotts Valley, CA 95066

3. "Programmer's Problem Solver for the IBM PC, XT & AT"
by Robert Jordain
A Brady Book
Published by Prentice Hall Press
New York, NY 10023
1986

XED4/5/8 Integrated Editor Cross-Assembler

XED4/5/8 is a fast and convenient method to develop and debug small to medium size programs. For use on Z80 machines running Z-system or CP/M. Companion **XDIS4/5/8** disassembler also available.

Targets: 8021, 8022, 8041, 8042, 8044, 8048, 8051, 8052. 8080, Z80, HD64180, and NS455 TMP.

Documentation: 100 page manual.

Features include:

★ Memory resident text (to about 40 KB) for very fast execution. Recognises Z-system's DIR: DU:. Program re-entry with text intact after exit.

★ Built in mnemonic symbols for all 8044,51,52 SFR and bit registers, NS455 TMP video registers and HD64180 I/O ports.

★ Output to disk in straight binary format. Provision to convert into Intel Hex file. Listing to video or printer. A sorted symbol table with value, location, all references to each symbol.

★ Supports most algebraic, logic, unary, and relational operators. Eight levels of conditional assembly. Labels to 31 significant characters.

★ A versatile built in line editor makes editing of individual lines, inserting, deleting text a breeze. Fast search for labels or strings. 20 function keys are user configurable.

★ Text files are loaded, appended, or written to disk in whole or part, any time, any file name. Switchable format to suit most other editors.

★ The assembler may be invoked during editing. Error correction on the fly during assembly, with detailed error and warning messages displayed.

For further information, contact:

PALMTECH
(a division of Palm Mechanical)
cnr. Moonah & Wills Sts.
BOULIA, QLD. 4829
Phone: 6177 463-109 Fax: 6177 463-198 AUSTRALIA

Data Structures in Forth

(Continued from page 11)

search the whole list for each symbol, only till you reach the point where the symbol should have been. Another advantage is that the symbol table does not have to be sorted into alphabetical order to print it out at the end of the assembly. This can be useful in many lists. You have to search the list anyway and you can save the sort time and space that would be needed to change the order of things later.

In the Forth example, the word $<12 is used to test if a 12 character string is less than another. The word $=12 tests to see if two 12 character strings are equal. The word newsym allocates space for a new symbol node.

No matter what language you choose, you should try to write good, understandable code. Try to keep things simple. Don't be lazy or try to save a couple of bytes or a few lines of code at the expense of being understandable. Design your code so it can be changed without falling apart. Spend some time and effort on finding good ways of doing what you are trying to do. After you write something, look at it again in a few days. Most people will not send out the first draft of a letter without reading it and trying to make it better, so do the same with your code. A little effort in this area will be noticed by your boss or your peers as well as save you effort figuring out your own code. It can pay off in many indirect ways. ●

LINKPRL

Making RSXes Easy

by Harold F. Bower

The first part of this article described a simple MicroSoft REL linker and detailed its operation. This linker, LINKPRL, produces either a standard COM file for execution under CP/M and compatible operating systems, or a "Page-Relocatable" PRL file suitable for making Resident System Extensions (RSXes) or ZCPR 3 Type 4 modules. In this second part of the article, both modes will be demonstrated using an example of a disk directory buffer RSX following the Plu*Perfect standard definition. Before proceeding to the detailed "how to" instructions for linking and forming the SPEEDUP RSX, as we shall call it, let us first examine the "how" of its operation.

SPEEDUP is a routine which caches part or all of a disk directory in memory, and accesses the image instead of the real disk directory for read operations. Writes to the cached disk update both the memory image and the actual disk directory to protect against data loss. The advantage of caching the directory is to obtain greater speed when reading files. For example, loading needed system files for ZCPR 3 becomes a lightning-fast series of reads, and reading large files is no longer punctuated by seeks to the directory when opening new extents. Memory for the cache is obtained from the Transient Program Area (TPA) in high memory, and will function under CP/M 2.2, ZRDOS 1.x and ZSDOS 1.x.

Architecture.

SPEEDUP is comprised of three logical segments grouped in two physical modules. The first module, SPEEDLDR.Z80, is a loader which performs the functions of validating certain system parameters, determining selected addresses to be passed to the relocated module, and performing needed address relocations for the Page Relocatable portion. All program-dependent parts of the loader, except for a couple of configuration options and built-in Help, are added to the second physical module. With this architecture, only minimal changes are needed to adapt SPEEDLDR to other RSX applications.

SPEEDLDR is linked to a .COM file for execution at 100H. It is exactly 1024 bytes (1K) long to make combination with the next module easier. The next few bytes after the end of the loader are reserved for the RSX header structure which appears at the beginning of the module to be relocated.

The second physical module, SPEED22.Z80, is relocated to high memory immediately below the Console Command Processor, or lowest existing RSX in its entirety. Two logical sections of code exist in SPEED22; a final installation section, and a resident core of code. Final installation consists of; determining the remaining addresses and values needed, patching the BIOS jump table, resident module and Page 0 addresses, and exiting via the resident module Warm Boot entry point. Installation is only needed once, so the code storage space in high memory is reclaimed for use by the resident portion of the RSX. Since both SPEEDLDR and the installation portion of SPEED22 are relatively self-explanatory, they will not be described in detail here, but may be obtained in source code form from at least the two Z-Nodes cited at the end of this article.

The final logical portion of code is the resident part of SPEED22 and is the heart of this RSX. Fewer than 650 bytes of code and data make up this segment. This functional division was selected to provide the maximum possible Transient Program Area and maximum re-usable source code for other projects. I hope you too find this structure applicable in RSX generation.

How SPEEDUP Works.

In order to understand how SPEEDUP works, you must understand the table-driven nature of CP/M and the interface between the Basic Disk Operating System (BDOS) which works on a logical file basis, and the Basic IO System (BIOS) which contains the physical device drivers. CP/M and many programs which access the BIOS directly for disk functions specify Track Number, Sector Number, DMA Transfer Address and Disk Selection before performing a physical Read or Write. The resident portion of SPEEDUP traps each of these BIOS entry points. A few of them (Set Track, Set Sector and Set DMA Transfer Address) merely store the specified values locally before passing them to the real BIOS. Other entry points are acted on locally and may not, in the case of a Sector Read, access the real BIOS at all. Listing 1 contains an extract of the SPEED22 source code and will be used to explain the inner workings of this RSX.

The BIOS Select Disk function is the first unit of SPEEDUP which takes local action. BIOS calls are routed to the RSX at label SELDSK where the disk number is compared against the one we specified when loading SPEEDUP. If the drive is not ours, we deselect the RSX and go directly to the real BIOS. If it is for us, we then check a cue flag which the BDOS provides as Bit 0 of the E-register. If this bit is Non-Zero (Set), the BDOS has already logged this disk and we again exit to the real BIOS just to keep everything synchronized. Only when a new mount request is received signified by this bit being reset (Zero) do we act locally by calling the real BIOS Disk Select function and returning. In response to a Select Disk function, BIOS returns a pointer to the 16-byte Disk Parameter Header (DPH) for the subject drive which contains a table of pointers to various drive parameters. Two of these, the address of the Skew Translation routine, and the address of another table—the Disk Parameter Block (DPB), are used by the RSX. The DPB provides us with the number of logical 128-byte Sectors Per Track on the drive, the number of Directory entries, and the number of reserved tracks on the disk before the the beginning of the Directory.

Values obtained from the BIOS Select Disk call are used to detect when a Read or Write request reference a Directory sector, and to determine whether the requested sector is in the memory buffer. When SPEEDUP is first loaded, it allocates memory in 1K increments as specified by calling parameters, with each 1K representing 32 directory entries. Actually, slightly more than that is allocated since one extra byte for the logical sector number is needed for every 4 Directory entries representing a logical sector. These logical sector numbers are placed in a table immediately below the resident module base, with sector storage extending downward from there. This sector number table incorporates any

needed skew by calling the Sector Translate routine obtained from the DPH as it is built. After the table is built, Directory sectors are read into memory until either the allocated memory is full, or the entire Directory is read. Execution then returns to the calling program, or the BDOS.

Requests for BIOS Writes and Reads are similarly vectored through the resident module at labels WRITE and READ respectively in the listing. Write requests, as with Select Disk calls, are also furnished a flag clue by the BDOS. In this case, the clue is the byte in the C-register which will be 01H if a Directory write is being requested. Any writes other than those for the Directory are simply routed directly to the real BIOS for processing. Those which involve the Directory first call the real BIOS, then check to see if the subject sector is in the memory cache (subroutine SETUP). If the Sector is present, the cache contents are also updated before returning to the calling program.

Read requests are not furnished with any special clues by the BDOS as in Write and Disk Select routines. A determination of whether or not the request involves cached sectors involves simply checking to see if the desired Track and Sector is in memory (subroutine SETUP). If not, we go directly to the real BIOS. If we have the sector cached, we merely move the sector to the location specified by the DMA address.

If you carefully think about the way this all works, you will detect the one pitfall in caching the Directory. A disk change without a Relog (Control-C or Dos functions 13 or 37) preceding a Write will, in all probability, trash the disk. Even the advanced Disk Change logic in ZSDOS will not detect a swap under these circumstances, so ALWAYS REMEMBER TO DO A WARM BOOT IMMEDIATELY AFTER CHANGING DISKS!

How To Make SPEEDUP.

Assuming that you have, by now, obtained LINKPRL from one of the available Z-Nodes, and the source code library for SPEEDUP, you are ready to form the executable RSX. First assemble both SPEEDLDR.Z80 and SPEED22.Z80 with M80, ZAS or any other assembler which produces a standard MicroSoft REL file. Next link SPEEDLDR with LINKMAP in the COM mode. The following interaction should be displayed where <cr> signifies the "Enter" or "Return" key on your keyboard.

```
>LINKPRL SPEEDLDR<cr>                          <-- Invoke LINKPRL

Bit-map Linker V3.2       13 Aug 89  (C) H.F.Bower

Link to .COM or .PRL (C/P) :C<cr>              <-- Link to COM
Enter Hex load addr (Default = 0100H) :<cr><-- Use Default
Program Name  : SPEEDL
Data Area Size   : 0000
Program Size  : 0400
```

If you examine your current directory at this point, you should have generated the file SPEEDLDR.COM. Next, we use LINKPRL to generate a PRL file for the segment which will be relocated to high memory. The following interaction should be displayed for this operation.

```
>LINKPRL SPEED22<cr>                           <-- Invoke LINKPRL

Bit-map Linker V3.2       13 Aug 89  (C) H.F.Bower

Link to .COM or .PRL (C/P) :P<cr>              <-- Link to PRL
Enter Hex load addr (Default = 0100H) :<cr><-- Use Default
Program Name  : SPEED2
Data Area Size   : 0000
Program Size  : 03D2
Load Location : 01B3

Bit Map begins @ ORG + 03D2
Bit Map ends @ ORG + 0444
```

As before, a check of your current directory will show that you have created SPEED22.PRL. See the first part of this article for a description of what special characteristics are exhibited by this type of file. For now, we merely want to use it. We do this by combining SPEEDLDR.COM and SPEED22.PRL into a single module called

Listing 1: An extract of the SPEED22 source code.

```
;*********************************************************
;*     M  O  D  U  L  E        B  O  D  Y               *
;*********************************************************
; Modified BIOS Select function loads buffer on New Login

SELDSK:  LD A,(LDISK)      ; Is it this drive ?
   CP C
   LD A,0       ; Deselect first...
   LD (FSEL),A
   JR NZ,BSDSKE   ; ..jump if not here
   DEC   A      ; Else select with 0FFH
   LD (FSEL),A
   BIT   0,E      ; Is it a logon request?
   JR NZ,BSDSKE   ; ..jump if no to BIOS
   CALL  BSDSKE            ; Else Do BIOS & return
   PUSH  HL     ; Save Regs for routine
   PUSH  BC
   PUSH  AF
   LD E,(HL)      ; Get skew addr
   INC   HL
   LD D,(HL)
   LD (SKWTBL),DE ; ..and save locally
   LD DE,9
   ADD   HL,DE    ; DPB is at DPH+10
   LD E,(HL)      ; ..and get DPB's address
   INC   HL
   LD D,(HL)
   EX DE,HL
   LD DE,SPTRK ; Move DPB to local area
   LD BC,15    ; ..all 15 bytes
   LDIR

   LD HL,(DIRMAX) ; Load # Dir entries - 1
   INC   HL     ; Correct dirmax

   SRL   H      ; Conv DIRMAX to # Sctrs.
   RR L     ; ..by dividing by 4
   SRL   H
   RR L
   EX DE,HL    ; Save in DE while we..
   LD HL,(DMAX)   ; ..compare to avail #
   LD H,0
   PUSH  HL    ; (preserve count)
   OR A
   SBC   HL,DE    ; Compare by subtraction
   POP   HL
   JR C,SMALR     ; Jump if this won't fit
   EX DE,HL    ; ..else save whole Dir
SMALR:   LD A,L   ; Get LS Byte
   LD (NDIRS),A   ; ..and save it
   LD DE,-1    ; Preset counter
   LD BC,(SPTRK)  ; ..get Sectors per Track
CALCLP:  INC   DE
   XOR   A
   SBC   HL,BC    ; How many trks for Dir?
   JR NC,CALCLP   ; Fall thru if DE = #Trks
   LD HL,(OFFSET) ; Get reserved trk count
   ADD   HL,DE    ; .add # of Dir tracks
   LD (TOPTRK),HL ; ..and save top Dir Trk
   LD HL,(SECTBL)
   LD A,(DMAX) ; Store this many entries
   LD B,A
   XOR   A     ; Fill table with Zero
CLRLP:   LD (HL),A
   INC   HL
   DJNZ  CLRLP    ; ..looping til done
   LD (CURSEC),A  ; Set Curr Log. Sctr to 0
   LD HL,(OFFSET)
   LD (TRK),HL ; Set starting Track

   LD HL,(SECTBL) ; Get Sctr ID Table start
   LD (TEMP),HL   ; ..and save

   LD HL,(DIR) ; Get start of Sctr Buff
   LD A,(NDIRS)   ; .# of Dir Sctrs therein
   LD B,A      ; ..to count register

; Load translated sector table and sector data buffer

SETTBL:  PUSH  BC          ; Save Cnt and Addr regs
   PUSH  HL
   LD DE,(SKWTBL) ; Set up for translation
   LD A,(CURSEC)  ; ..on current sector
   LD C,A
   LD B,0
   INC   A     ; Bump Log. Sctr number
   LD (CURSEC),A
   CALL  BSCTRN            ; Find translated sector
   LD A,L
   LD HL,(TEMP)
```

```
   LD (HL),A      ; Store translated sector
   INC   HL     ; ..and bump table Addr
   LD (TEMP),HL
   LD C,A
   CALL  BSETSC            ; Set Sector Number
   POP   BC     ; Retrieve Buffer Address
   PUSH  BC     ; ..and keep on stack
   CALL  BSTDMA            ; DMA addr to Buffer Sctr
   LD BC,(TRK)
   CALL  BSTTRK            ; Set Physical Trk Number
   CALL  BREAD     ; ..and read Disk Sector

   POP   HL     ; Restore registers
   POP   BC
   LD DE,128       ; Increment cache address
   ADD   HL,DE
   LD A,(CURSEC)
   LD DE,(SPTRK)
   CP E      ; Check Track overflow
   JR C,SETTB0 ; ..jump if same Track
   LD DE,(TRK)
   INC   DE     ; Else bump track #..
   LD (TRK),DE
   XOR   A      ; ..and zero Sctr Number
   LD (CURSEC),A
SETTB0:  DJNZ  SETTBL
   JR WEXIT     ; Restore parms & Exit

;.....
; BIOS Write Sector routine.  If sector resident,
;  update memory image in addition to disk data

WRITE:   LD A,C
   DEC   A      ; directory write?
   JP NZ,BWRITE   ; .jump if not to BIOS
   CALL  BWRITE            ; ..else write to disk
   PUSH  HL     ; save status
   PUSH  BC
   PUSH  AF
   CALL  SETUP     ; Is this Trk/Sec in RAM?
   JR NZ,WEXIT ; ..quit if not
   OR 0FFH      ; Show this is a Write
   JR RAMW      ; ..and update Cache

;.....
; BIOS Read sector routine.  If sector in memory, simply
; move, else jump to real BIOS and get data from disk

READ: LD DE,(TRK) ; Directory track?
   LD HL,(TOPTRK) ; See if trk <= toptrk
   OR A
   SBC   HL,DE     ; Compare by subtraction
   JP C,BREAD      ; ..if not dir, read disk
   CALL  SETUP     ; Else is Trk/Sec in RAM?
   JP NZ,BREAD ; ..jump to Disk if Not
   LD HL,0      ; Set "good read" status
   PUSH  HL     ; .once for HL
   PUSH  HL     ; ..and once for BC
   XOR   A      ; Show this is a Read
   PUSH  AF     ; ..with good status

; Calculate address of desired sector in RAM buffer
;  ENTRY: Reg C has relative sector number
;  EXIT:  HL register pair has buffer absolute address

RAMW: PUSH  AF     ; Save flag status
   PUSH  DE
   LD D,C       ; Use Sctr # as index
   LD E,0       ; .into RAM Cache
   SRL   D      ; Multiply index by 128
   RR E
   LD HL,(DIR) ; Get base of Cache
   ADD   HL,DE     ; ..and add offset
   POP   DE
   POP   AF

   LD DE,(DMADDR) ; Data transfers go here
   OR A      ; Check which operation
   JR Z,RAM0       ; ..jump if read
   EX DE,HL     ; Swap direction if Write
RAM0: LD BC,128            ; Move a logical Sector
   LDIR
WEXIT:   POP   AF          ; Restore status & Regs
   POP   BC
   POP   HL
   RET

;-------------------------------------------------------;
;  S U P P O R T       R O U T I N E S                  ;
;-------------------------------------------------------;
; Check for presence of requested sector in memory buffer
;  ENTRY: None
;  EXIT: Found - Zero flag set
;           C has relative 128-byte Sctr offset
;        Not Found - Zero flag cleared (reset)
;          Reg A is Non-Zero

SETUP:   LD A,(FSEL)       ; Is this drive selected?
   OR A
   JR Z,SETUP8 ; ..error return if not
   LD BC,(OFFSET) ; Calc # tracks from 1st
   LD HL,(TRK)
   XOR   A      ; Get offset from Direc.
   SBC   HL,BC
   LD A,L       ; ..should be < 255
   LD HL,0      ; Make # of Sctrs offset
   LD DE,(SPTRK)  ; Get # of Sectors/Track
SETUP2:  OR A
   JR Z,SETUP3
   ADD   HL,DE     ; .adding to base
   DEC   A
   JR SETUP2       ; ..loop til finished

SETUP3:  LD C,L   ; Copy # sectors to BC
   LD B,H
   EX DE,HL     ; ..and store in DE
   LD HL,(NDIRS)  ; Get # Dir Sctrs stored
   LD H,0
   XOR   A      ; Subtract offset sectors
   SBC   HL,BC
   JR C,SETUP8 ; Not Found if overflow
   LD B,L       ; Put 8-bit Sctr cnt in B
            ; ..Sector offset is in C
   LD HL,(SECTBL) ; Get addr fm table start
   ADD   HL,DE
   LD A,(SCTR+1)  ; Is Sctr # > 256?
   OR A
   JR NZ,SETUP8   ; ..not here if so
   LD A,(SCTR) ; See if sectors match
RL1:  CP (HL)
   JR Z,SETUP9 ; .jump if found
   INC   C
   INC   HL
   DJNZ  RL1      ; ..else loop til done
SETUP8:  DEFB  0F6H                ; Set Error w/"OR 0AFH"
SETUP9:  XOR   A           ; Return Ok conditions
   RET

;---------------------------------------;
;  D A T A      A R E A                 ;
;---------------------------------------;
; Storage for data passed to BIOS

SCTR: DEFS  2      ; Sector passed to BIOS
TRK:  DEFS  2      ; Track passed to BIOS
DMADDR:  DEFS  2           ; DMA Addr passed to BIOS

; DPB and Skew Table address for active disk

SPTRK:   DEFS  2           ; Sectors per track
   DEFS  5      ;  Room for BSH, BLM etc
DIRMAX:  DEFS  2           ; Max # of Dir entries
   DEFS  4      ;  Room for CHK & ALV
OFFSET:  DEFS  2           ; Track offset

SKWTBL:  DEFS  2           ; Skew table address

; General Purpose pointer and buffer area

FSEL: DEFS  1      ; drive selected flag
LDISK:   DEFS  1           ; Disk # active in RAM
DIR:  DEFS  2      ; Start of Dir Buffer RAM
SECTBL:  DEFS  2           ; Addr of Sector #s Table
TOPTRK:  DEFS  2           ; Top track of Directory

CURSEC:  DEFS  1           ; Current working sector
DMAX: DEFS  1      ; Maximum # Dir sectors
NDIRS:   DEFS  1           ; # of DIR Sectors stored
TEMP: DEFS  2      ; Temp pointer area

   DEFS  32     ; Local stack space
STACK:   DEFS  2           ; Storage for entry stack

   END
```

(Continued on page 31)

Real Computing

The National Semiconductor NS32032

by Richard Rodman

I'm going to imitate Jay Sage this time and write about something other than what I said I was going to write about.

Hobby computing started out as an exciting pastime for computer professionals because the early machines were simple and easy to understand. You could know the whole machine. When you wrote data to an I/O port, there were no Session and Transport layers. These pioneers (arrows in their backs and all) brought many newcomers into computing, and they got used to having that kind of simple interface and complete control. But when PCs became "legitimized" by the latecomer IBM, they lost their innocent simplicity and headed down the road towards the bulky, inefficient traits of the Computing Status Quo.

Similarly, RISC started out as a clumsy effort to regain simplicity and cleanness of design. When it went commercial, it soon lost its direction, because it had to be "dressed up" to meet the requirements of all of the old stuff.

But as hardware designers have run into stiffening limits to improving performance, more and more people have wondered where in the world all of these MIPS and megabytes per second are going, and found that 40 percent or more are going into a black hole called "Overhead". That overhead is why a little Z-80 could do things faster than a mighty VAX or a mainframe.

That is why the computing world will soon be hit like a lightning bolt by a new design philosophy, which I refer to as *Low-Overhead Systems* (LOS). LOS designers will strive, not to minimize the overhead or shift it into hardware, but to eliminate it altogether. The OSI 8-layer model will wane in favor of a new "connectionless" 3-layer model.

Imagine the simplicity of CP/M being moved to an IBM mainframe. Just think of the MIPS that'll be freed. This is the shared vision of simple beauty that made the hobby computer, that typifies many experimenters of today, and that typifies *TCJ* readers. We've been designing low-overhead systems all along.

Free OS Update

Version 0.4 of Bare Metal is released now and is available from this magazine. This is an intermediate release, of course, and is not a complete system yet. However, if you've got some NS32 hardware that you want to run programs on, it should be useful to you.

Version 0.4 supports two devices, a remote device, REM:, and a MS-DOS-compatible 360K drive, DOS:. Programs may be run from either device. The REM: device is for serial-link or coprocessor board access. The HOSTX program, which otherwise is a simple terminal program with download capability, has been extended to support this device.

Metal has a really crummy built-in command interpreter. It currently offers the following commands: DIR, TYPE filename, ERASE filename, and COPY filename1 filename2.

Programs to be loaded must have the standard 16-byte header used in the SRM monitor. This header is based on the NS32 Module Table entry. The first doubleword contains the BSS size, that is, how much uninitialized memory should be prepended to the initialized data to form the program's data area, which is pointed to by the SB register. The second doubleword contains the offset to the link table for the module. The third doubleword contains the offset to the code for the module. The fourth doubleword contains the total size of the module, which includes the BSS size. After this comes the initialized data and then the code.

Most importantly, the program must be relocatable without modification—it cannot contain any address constants (such as initialized pointers). This is a key concept of Metal. On the one hand, this "breaks code"—C programs which would work fine in other OSs have to be modified. On the other hand, it significantly quickens program loading and will allow Metal to support multitasking without an MMU.

OS Calls

The operating system calls shown in Figure 1 are available from Metal. (By the way, keep this article, this is the best documentation there is right now!)

The value of "handle" in the calls might be 0, for stdin/stdout, or 3 and up for files. Handle 1 is reserved for Aux, and handle 2 for the printer.

To perform the operation, load the registers as shown and perform an SVC instruction.

BIOS Entry Points

The BIOS for Metal may be written in C if desired. The entry points are defined in terms of C functions. The definitions shown here may change in future versions.

void coldboot(void); initializes everything used by the BIOS.

void consout(char); sends a character out to the console. An assembler routine will find the character at 4(SP).

int consinp(void); reads a character from the console. It waits for the character to be available. The character should be returned in R0.

int writesec(int sectornumber; char *ioaddr); writes a sector

Figure 1: Metal operating system calls.

R0	R1	R2	R3	
--	--	--	--	
1	-	-	handle	getc - Read a character
2	char	-	handle	putc - Write a character
3	buffer	count	handle	read - Read count characters (Disk not implemented yet)
4	buffer	count	handle	write - Write count characters (Disk not implemented yet)
5	filename	-	-	open - Open a file
6	-	-	handle	close - Close a file
7	filename	-	-	create - Create a file
10	filename	-	-	erase - Erase a file
11	oldname	newname	-	rename - Rename a file
21	dirname	-	-	mkdir - Create a directory
25	dirname	-	-	chdir - Change directory

(512 bytes) from memory address ioaddr to the disk absolute sector sectornumber. This call's usage is reminiscent of the MS-DOS int 25. The value returned in R0 should be 0 for success or some negative number for failure. An assembly routine will receive sectornumber at 4(SP) and ioaddr at 8(SP).

int readsec(int sectornumber; char *ioaddr); is like writesec but reads the sector instead.

The sample BIOS distributed with Metal is written in assembler for the Cromemco 16FDC board, which has a floppy controller and a serial port on it.

Compiling and Assembling

Metal is about half C and half assembler. The current system generation method is rather clumsy. First, the C source file is preprocessed with the Decus C Preprocessor and compiled with Phil Prendeville's C compiler. Then, the assembly output of the C compiler must be edited to include the assembly language routines.

At the beginning of the file, you have to include, first, the file METSTART, then your customized BIOS. Then, at the end of the file, you have to include the file METMEM.

One other pain is that the C compiler tends to generate non-relocatable instructions of the form ADDD #label,R0. If label is an address, this sequence needs to be replaced with a two-line sequence ADDR label,R1 followed by ADDD R1,R0.

The resulting enormous file is assembled with A32. I suggest using the nobyte option, which prevents the branch optimization from using the 1-byte short offsets, in order to save assembly time.

The output from this is a .E32 file which can be loaded with SRM or, with some tinkering, TDS. Currently, that is the only way to load Metal.

Future Plans for the BIOS

In version 0.5, multiple drives per device type will be supported. There will be a table which will allow you to re-use a manager routine, such as the MS-DOS file manager, for multiple different drives, each of which will have a driver routines. This will allow you to mix and match managers and drivers independently. This will also remove the 360K format restriction.

Version 0.5 will also include drivers for the PD32 coprocessor board and possibly the 532 Designer's Kit board.

Booting, of course, is a problem. The MS-DOS boot procedure is quite complex, with all the various formats in use now. I'll entertain suggestions from readers on this one.

Metal handles nearly all of the NS32 traps now. In the future, it will incorporate program tracing and debugging right from the command line.

Next Time

Next time we return you to your normally scheduled information, where I promised to cover the NS32 trap mechanism and built-in single step mechanism. •

Where to write or call:

Richard Rodman
8329 Ivy Glen Court
Manassas VA 22110
BBS: 703-330-9049

The METAL Version 0.4 Operating System on a 360K MS-DOS disk is available from TCJ for $12.

LINKPRL

(Continued from page 29)

SPEEDUP.COM. The simplest approach is to use DDT, ZDMH,

```
>ZDMH SPEEDLDR.COM<cr>              <-- Load SPEEDLDR at 100H

ZDMH VERS 1.2
NEXT  PC
0500 0100
-ISPEED22.PRL<cr>                   <-- Load SPEED22.PRL
-R400<cr>        <-- ..to 500H (100H+400H)
NEXT  PC
0A80 0100
-G0<cr>        <-- Return to CP/M, image still
                   in memory
>SAVE 10 SPEEDUP.COM<cr>  <-- Save 100H to 0AFFH to Disk
```

ZBUG or any similar debugger. The following interaction will occur with ZDMH.

You now have an executable SPEEDUP.COM file in your current directory. Operation is quite simple with a brief help message covering usage available which may be displayed by entering:

SPEEDUP //

I hope you found this article to be of some help in understanding the finer points of PRL files, and will be able to use these tools. Complete source code for the programs described here may be obtained in LINKPRL.LBR and SPEEDUP.LBR from the Ladera Z-Node operated by Al Hawley at (213) 670-9465 and from Jay Sage's Z-Node #3 at (617) 965-7259. I thank both Al and Jay for allowing me to furnish these programs on their systems. Any comments/bugs etc. for me may be addressed to:

Harold F. Bower
P.O. Box 313
Ft. Meade, MD 20755

Correction for issue #40

The LINKPRL article in issue #40 contained an error on page 18. The fourth line from the bottom in the left hand column should read "...the Code area is placed in the second and third bytes..."

SCOPY

Selective Duplication of MS-DOS Files

by Dr. Edwin Thall

The MS-DOS COPY command, with its wildcard capabilities, is adept at duplicating file names with similar characters. Sometimes, however, it's impossible to find a wildcard combination to solely duplicate selected files. Suppose, for example, you have six files with the same name (TEST) but different extensions (.TXT, .BAK, .DOC, .EXE, .ASC, .MSG) and you wish to copy the first three files from drive A to drive B. All files are copied if you enter:

```
A>COPY TEST.* B:
```

To prevent duplication of the unwanted files, you need to type three separate commands:

```
A>COPY TEST.TXT B:
A>COPY TEST.BAK B:
A>COPY TEST.DOC B:
```

Recently, I was asked about the feasibility of a utility that can selectively copy files from one directory to another without necessitating the typing of file names. After acknowledging that such a program would not be difficult to write, the challenge was born. Although the project turned out to be more time-consuming than first anticipated, I did create SCOPY—for selective copy. But before introducing SCOPY, let's examine how files are actually duplicated.

From Handles to Z-Strings

Files are readily created and accessed by means of the DOS functions listed in Table 1. All of these functions, designated "H" for handle and "A" for ASCIIZ string, have been incorporated into SCOPY.

The ASCIIZ format consists of a series of conventional ASCII characters terminated by a byte of zero (00H). For the ASCIIZ string shown below:

```
'A:\PATH\NAME.EXT',0
```

The drive (A:), path (\PATH), and file name (\NAME.EXT) are stored in memory as:

```
41 3A 5C 50 41 54 48 5C 4E 41 4D 45 2E 45 58 54 00
```

The null byte (00H) cannot be entered directly by way of the keyboard since keystrokes such as zero or Alt<0> return 30H and nothing, respectively. The backslash (\) serves as path separators.

Whenever you create or open a file (functions 3CH/3DH), the name of the file is specified with an ASCIIZ string. DOS maintains the file's control information in its own area, and returns a number in the AX register. This number, known as the file handle, must be referred to for future access of the file.

DOS relies on file handles to keep track of files and input/output devices. The number of files that DOS can open concurrently may be declared during the boot with CONFIG.SYS (FILES = N). A maximum of 20 handles is permitted with the default value set at 8.

File Duplication

Without benefit of the COPY command, let's duplicate a single file from the root directory to a target subdirectory. Initially, the source file is read into a buffer area. Next, a new file with the same name as the source is created in target directory . The buffer area is then written to the new file in the target directory.

We begin by creating a source file (A:\SOURCE) in the root directory of drive A. The message "FILE DUPLICATION" will be the only data stored in this file. From drive A, enter:

```
A>COPY CON A:\SOURCE
FILE DUPLICATION
Ctrl Z
```

The execution of this file displays the message "FILE DUPLICATION" on the screen:

```
A>SOURCE
FILE DUPLICATION
```

Before attempting to duplicate this file, invoke the DOS command to create the subdirectory \TARGET:

```
A>MD\TARGET
```

The duplication of our source file as A:\TARGET\SOURCE is accomplished with DUP.COM, assembly code listed in Figure 1. DUP.COM is quite limited and serves only one purpose; it copies SOURCE from the root directory to the newly formed subdirec-

Function	Operation	
3BH	Create file	(A)
3DH	Open file	(A)
3EH	Close file	(H)
3FH	Read file or device	(H)
40H	Write to file or device	(H)
42H	Reset file pointer	(H)
4EH	Search first match	(A)
4FH	Search next match	(A)

Table 1. Selected DOS Functions (A=ASCIIZ string, H=handle)

Bytes	Represents
0-20	Reserved by DOS
21	Attribute of matched file
22-23	Time
24-25	Date
26-29	Size
30-42	File name/extension (ASCIIZ)

Table 2. Information returned in the DTA by functions 4EH/4FH

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

```
;DUP.ASM converts to DUP.COM
;Duplicates A:\SOURCE as A:\TARGET\SOURCE
CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG
        ORG  100H
START:  JMP  SKIP                   ;skip data
SOURCE  DB   'A:\SOURCE',0          ;asciiz--source
TARGET  DB   'A:\TARGET\SOURCE',0   ;asciiz--target
SHANDLE DW   ?                      ;file handle--source
THANDLE DW   ?                      ;file handle--target
BUFFER  DB   16 DUP (''B'')           ;16 byte buffer
;open source file
SKIP:   MOV   AH,3DH                ;open file
        MOV   AL,0                  ;read operation
        MOV   DX,OFFSET SOURCE      ;source asciiz
        INT   21H
        MOV   SHANDLE,AX            ;store source handle
;create target file
        MOV   AH,3CH                ;create file
        MOV   CX,0                  ;normal file
        MOV   DX,OFFSET TARGET      ;target asciiz
        INT   21H
        MOV   THANDLE,AX            ;store target handle
;read source file into buffer
        MOV   AH,42H                ;reset file pointer
        MOV   AL,0                  ;from start of file
        MOV   CX,0                  ;set pointer high
        MOV   DX,0                  ;set pointer low
        MOV   BX,SHANDLE            ;get source handle
        INT   21H
        MOV   AH,3FH                ;read file
        MOV   CX,16                 ;number of bytes
        MOV   DX,OFFSET BUFFER      ;point to buffer
        INT   21H
;write buffer to target file
        MOV   AH,42H                ;reset file pointer
        MOV   AL,0                  ;from start of file
        MOV   CX,0                  ;file pointer high
        MOV   DX,0                  ;file pointer low
        MOV   BX,THANDLE            ;get target handle
        INT   21H
        MOV   AH,40H                ;write to file
        MOV   CX,16                 ;number of bytes
        MOV   DX,OFFSET BUFFER      ;point to buffer
        INT   21H
;close files
        MOV   AH,3EH                ;close file
        MOV   BX,SHANDLE            ;get source handle
        INT   21H
        MOV   AH,3EH
        MOV   BX,THANDLE            ;get target handle
        INT   21H
        INT   20H
CSEG    ENDS
        END   START
```

Figure 1. Assembler code for DUP.COM

tory. Here is how DUP.COM works:

1. The source file is opened (function 3DH) and its handle saved in the area defined by SHANDLE.
2. A new file (A:\TARGET\SOURCE) is created and its handle saved in the location specified by THANDLE.
3. The 16 bytes of data (FILE DUPLICATION) stored in A:\SOURCE are read into a buffer area (function 3FH).
4. The file pointer is reset (function 42H) to the beginning of the file.
5. The data is written (function 40H) from the buffer area to A:\TARGET\SOURCE.
6. Both files are closed (function 3EH).

Execute DUP.COM and then verify the existence of A:\TARGET\SOURCE:

```
A>DUP
A>TYPE \TARGET\SOURCE
FILE DUPLICATION
```

Figure 2: Assembler code for SCOPY.EXE

```
;SCOPY.ASM converts to SCOPY.EXE
;Selectively copies files from one directory to another
;*********************************************************
SSEG    SEGMENT  STACK
        DB       20 DUP  ('STACK   ')
SSEG    ENDS
;*********************************************************
DSEG    SEGMENT
SOURCE  DB   64,65 DUP (0)          ;asciiz--source dir
SOURCE2 DB   65 DUP (0)             ;asciiz--source filename
SOURCE3 DB   65 DUP (0)             ;asciiz--source current dir
TARGET  DB   64,65 DUP (0)          ;asciiz--target dir
TARGET3 DB   65 DUP (0)             ;asciiz--target current dir
DTA     DB   43 DUP (0)             ;search function DTA
SENTRY  DW   ?                      ;file entry into source2
TENTRY  DW   ?                      ;file entry into target
SETDIR  DB   0,':\',65 DUP(0)       ;store original dir
SHANDLE DW   ?                      ;source handle
THANDLE DW   ?                      ;target handle
REM     DW   ?                      ;remainder beyond 32K
BSIZE   DW   ?                      ;bytes to read/write
POINTSH DW   0                      ;source pointer high
POINTSL DW   0                      ;  ''       ''    low
POINTTH DW   0                      ;target pointer high
POINTTL DW   0                      ;  ''       ''    low
DISKF   DB   0
MESS1   DB   0AH,0AH,0DH,'ENTER SOURCE DIRECTORY:',0AH,0DH
        DB   '(EXAMPLES:  A:\, B:\PATH\, C:\PATH1\PATH2\)'
        DB   0DH,0AH,0AH,24H
MESST   DB   0AH,0AH,0DH,'ENTER TARGET DIRECTORY:',0AH,0DH
        DB   '(EXAMPLES:  A:\, B:\PATH\, C:\PATH1\PATH2\)'
        DB   0DH,0AH,0AH,24H
MESS2   DB   0DH,0AH,'COPY FILE?    '
        DB   '<Y> YES   <N> NO   <Q> QUIT    $'
MESS3   DB   0DH,0AH,'FILE COPIED  $'
MESS4   DB   0DH,0AH,'FILE COULD NOT BE COPIED $'
MESS5   DB   0DH,0AH,0AH,'NO FILES LOCATED',0DH,0AH,0AH,24H
MESS6   DB   0DH,0AH,'DISK FULL',0DH,0AH,0AH,24H
CRLF    DB   0DH,0AH,0AH,24H
BUFFER  DB   8000H DUP (''B'')        ;32K buffer
DSEG    ENDS
;*********************************************************
CSEG    SEGMENT
MAIN    PROC   FAR
        ASSUME CS:CSEG,DS:DSEG,ES:DSEG,SS:SSEG
START:
;set return & DS/ES registers
        PUSH DS
        SUB  AX,AX
        PUSH AX
        MOV  AX,DSEG
        MOV  DS,AX
        MOV  ES,AX
;call subroutines
        CALL SDIR           ;save original dir
        CALL CLEAR          ;clear screen
        CALL STRINGS        ;get dir & asciiz strings
        CALL SEARCH         ;search dir & copy option
        CALL RDIR           ;restore original dir
        RET
MAIN    ENDP
;---------------------------------------------------------
;save original directory
SDIR    PROC  NEAR
        MOV   AH,19H                ;get drive
        INT   21H
        ADD   AL,41H                ;declare drive as A,B,C,D
        MOV   SETDIR,AL             ;save drive
        MOV   AH,47H                ;get dir path
        MOV   DL,0                  ;default drive
        MOV   SI,OFFSET SETDIR+3    ;store path
        INT   21H
        RET
SDIR    ENDP
;---------------------------------------------------------
;restore original directory
RDIR    PROC  NEAR
        MOV   AH,3BH                ;set dir
        MOV   DX,OFFSET SETDIR
        INT   21H
        RET
RDIR    ENDP
;---------------------------------------------------------
;clear screen
CLEAR   PROC NEAR
        MOV  CX,25
CLR:    MOV  DL,0AH
        MOV  AH,2
```

```
        INT  21H
        LOOP CLR
        MOV  AH,2
        MOV  BH,0
        MOV  DX,0
        INT  10H
        RET
CLEAR   ENDP
;-------------------------------------------------------------
;input directories and set up asciiz strings
STRINGS PROC NEAR
;**source directory operations**
        MOV  AH,9                ;display message to input
        MOV  DX,OFFSET MESS1     ;source directory
        INT  21H
        MOV  AH,0AH              ;input
        MOV  DX,OFFSET SOURCE    ;source dir
        INT  21H
;set up asciiz as SOURCE2
        MOV   SI,OFFSET SOURCE+2 ;move string from SOURCE+2
        MOV   DI,OFFSET SOURCE2  ;to SOURCE2
        MOV   BX,OFFSET SOURCE+1 ;string length
        MOV   CL,[BX]
        MOV   CH,0
        CLD
        REP   MOVSB              ;move string
        MOV   SENTRY,DI          ;save entry into SOURCE2
;set up asciiz as SOURCE3 for current directory
        MOV   SI,OFFSET SOURCE+2
        MOV   DI,OFFSET SOURCE3
        MOV   BX,OFFSET SOURCE+1
        MOV   CL,[BX]
        MOV   CH,0
        CLD
        REP   MOVSB
        DEC   DI
        MOV   AL,0
        MOV   [DI],AL
;store *.* and null byte at end of SOURCE
        MOV  BX,OFFSET SOURCE+1  ;point to SOURCE size
        MOV  AL,[BX]             ;get size
        MOV  AH,0
        ADD  BX,AX               ;point to end of SOURCE
        INC  BX                  ;point to CR (0DH)
        MOV  AL,''*''
        MOV  [BX],AL
        INC  BX
        MOV  AL,''.''
        MOV  [BX],AL
        INC  BX
        MOV  AL,''*''
        MOV  [BX],AL
        INC  BX
        MOV  AL,0
        MOV  [BX],AL             ;store null byte
;**target directory operations**
        MOV  AH,9                ;display message to input
        MOV  DX,OFFSET MESST     ;input target directory
        INT  21H
        MOV  AH,0AH              ;input
        MOV  DX,OFFSET TARGET    ;target directory
        INT  21H
;set up asciiz as TARGET3 for current directory
        MOV   SI,OFFSET TARGET+2
        MOV   DI,OFFSET TARGET3
        MOV   BX,OFFSET TARGET+1
        MOV   CL,[BX]
        MOV   CH,0
        CLD
        REP   MOVSB
        DEC   DI
        MOV   AL,0
        MOV   [DI],AL
;get entry point into TARGET
        MOV  BX,OFFSET TARGET+1  ;point to TARGET size
        MOV  AL,[BX]             ;get size
        MOV  AH,0
        ADD  BX,AX               ;point to end of TARGET
        INC  BX                  ;point to CR (0DH)
        MOV  TENTRY,BX           ;save point into TARGET
        RET
STRINGS ENDP
;-------------------------------------------------------------
;search dir for match & offer option to copy file
SEARCH  PROC NEAR
        MOV  AH,1AH              ;set up DTA
        MOV  DX,OFFSET DTA
        INT  21H
        MOV  AH,4EH              ;search first match
        MOV  CX,20H              ;normal file
        MOV  DX,OFFSET SOURCE+2  ;point to source dir
        INT  21H
        JNC  MATCH               ;jump if match found
        MOV  AH,9                ;display message
        MOV  DX,OFFSET MESS5     ;''NO FILES LOCATED''
        INT  21H
        RET                      ;terminate program
NEXT:   MOV  AH,4FH              ;search nest match
        MOV  CX,20H              ;normal file
        MOV  DX,OFFSET SOURCE+2
        INT  21H
        JNC  MATCH               ;jump if match found
        RET                      ;terminate program
;blank out previous entry in SOURCE2
MATCH:  MOV  BX,SENTRY           ;entry into SOURCE2
        MOV  CX,13               ;replace 13 bytes
        MOV  AL,'' ''              ;with blank
BLANK:  MOV  [BX],AL
        INC  BX
        LOOP BLANK               ;blank next byte
;blank out previous entry in TARGET
        MOV  BX,TENTRY           ;entry into TARGET
        MOV  CX,13
        MOV  AL,'' ''
BLANK2: MOV  [BX],AL
        INC  BX
        LOOP BLANK2
;move file name to SOURCE2
        MOV  DI,SENTRY           ;entry into SOURCE2
        MOV  SI,OFFSET DTA+30    ;point to file name in DTA
        MOV  CX,13               ;13 byte asciiz string
        CLD
        REP  MOVSB               ;move from DTA to SOURCE2
;move file name to TARGET
        MOV  DI,TENTRY           ;entry into TARGET
        MOV  SI,OFFSET DTA+30    ;point to file name in DTA
        MOV  CX,13               ;13 byte asciiz string
        CLD
        REP  MOVSB               ;move from DTA to TARGET
        MOV  AH,9                ;skip line
        MOV  DX,OFFSET CRLF
        INT  21H
;display file name
        MOV  BX,OFFSET DTA+30    ;point to asciiz in DTA
SCREEN: MOV  DL,[BX]             ;check asciiz string for
        CMP  DL,0                ;null byte
        JE   OPTION              ;jump if end of asciiz
        MOV  AH,2                ;display character
        INT  21H
        INC  BX                  ;next asciiz location
        JMP  SCREEN
;option to copy file
OPTION: MOV  AH,9                ;display option message
        MOV  DX,OFFSET MESS2     ;''COPY FILE?''
        INT  21H
        MOV  AH,1                ;input Y, N, or Q
        INT  21H
        CMP  AL,''Y''
        JZ   COPY                ;if ''Y'', copy file
        CMP  AL,''y''
        JZ   COPY                ;if ''y'', copy file
        CMP  AL,''N''
        JZ   NEXT                ;if ''N'', next file
        CMP  AL,''n''
        JZ   NEXT                ;if ''n'', next file
        CMP  AL,''Q''
        JZ   QUIT                ;if ''Q'', quit program
        CMP  AL,''q''
        JZ   QUIT                ;if ''q'', quit program
        JMP  OPTION              ;repeat operation
COPY:   CALL COPYF               ;copy the file
        CMP  DISKF,0FFH          ;check if disk full
        JE   QUIT                ;if full, quit program
        JMP  NEXT                ;process next file
QUIT:   RET                      ;terminate program
SEARCH  ENDP
;-------------------------------------------------------------
;copy file from source to target
COPYF   PROC  NEAR
        CALL  SSDIR              ;get source dir
        MOV   AH,3DH             ;open source file
        MOV   AL,0               ;read operation
        MOV   DX,OFFSET SOURCE2
        INT   21H
        MOV   SHANDLE,AX         ;save source handle
        CALL  STDIR              ;set target dir
        MOV   AH,3CH             ;create file
```

```
        MOV   CL,DTA+21           ;attribute
        MOV   CH,0                ;''
        MOV   DX,OFFSET TARGET+2
        INT   21H
        MOV   THANDLE,AX          ;save target handle
;initialize file pointers
        MOV   POINTSL,0           ;source low
        MOV   POINTSH,0           ;source high
        MOV   POINTTL,0           ;target low
        MOV   POINTTH,0           ;target high
;divide file size by 32K
        MOV   DL,DTA+28           ;high order file size
        MOV   DH,DTA+29           ;''
        MOV   AL,DTA+26           ;low order file size
        MOV   AH,DTA+27           ;''
        MOV   CX,8000H            ;32K divisor
        DIV   CX
        MOV   REM,DX              ;save remainder
        MOV   CX,AX               ;count for 32K units
        CMP   REM,0               ;if rem = 0, do not
        JZ    AGAIN               ;increment counter
        INC   CX                  ;add 1 for rem
AGAIN:  PUSH  CX                  ;save count
        CMP   CX,1                ;rem?
        JNZ   K32                 ;process 32K
        MOV   DX,REM              ;read/write rem
        MOV   BSIZE,DX            ;store current buffer size
        JMP   SREAD               ;read source
K32:    MOV   DX,8000H            ;read/write 32K
        MOV   BSIZE,DX            ;store current buffer size
SREAD:  CALL  READ                ;read source to buffer
        JC    CFAIL               ;jump if read failed
        ADD   POINTSL,8000H       ;adjust source pointer
        JNC   NC1                 ;''
        ADD   POINTSH,1           ;''
NC1:    CALL  WRITE               ;write to target file
        CMP   AX,0                ;if disk full,
        JZ    FULL                ;quit program
        JC    CFAIL               ;if write failed, quit
        ADD   POINTTL,8000H       ;adjust target pointer
        JNC   NC2                 ;''
        ADD   POINTTH,1           ;''
NC2:    POP   CX                  ;restore count
        LOOP  AGAIN
        MOV   AH,9                ;display message
        MOV   DX,OFFSET MESS3     ;''FILE COPIED''
        INT   21H
        JMP   CLOSE
CFAIL:  POP   CX                  ;clean stack
        MOV   AH,9                ;display message
        MOV   DX,OFFSET MESS4     ;''FILE NOT COPIED''
        INT   21H
CLOSE:  CALL  SSDIR               ;set source dir
        MOV   AH,3EH              ;close source file
        MOV   BX,SHANDLE
        INT   21H
        CALL  STDIR               ;set target dir
        MOV   AH,3EH              ;close target file
        MOV   BX,THANDLE
        INT   21H
        RET
FULL:   POP   CX                  ;clean stack
        MOV   AH,9                ;display message
        MOV   DX,OFFSET MESS6     ;''DISK FULL''
        INT   21H
        MOV   DISKF,0FFH          ;disk full indicator
        RET
COPYF   ENDP
;-------------------------------------------------------------
;set source as current directory
SSDIR   PROC  NEAR
        MOV   AH,3BH              ;set current dir
        MOV   DX,OFFSET SOURCE3
        INT   21H
        RET
SSDIR   ENDP
;-------------------------------------------------------------
;set target as current directory
STDIR   PROC  NEAR
        MOV   AH,3BH              ;set current dir
        MOV   DX,OFFSET TARGET3
        INT   21H
        RET
STDIR   ENDP
;-------------------------------------------------------------
;read source file
READ    PROC  NEAR
        CALL  SSDIR               ;set source dir
        MOV   AH,42H              ;reset file pointer
        MOV   AL,0                ;from start of file
        MOV   CX,POINTSH          ;point source high
        MOV   DX,POINTSL          ;point source low
        MOV   BX,SHANDLE          ;get source handle
        INT   21H
        MOV   AH,3FH              ;read file
        MOV   BX,SHANDLE
        MOV   CX,BSIZE            ;No. of bytes
        MOV   DX,OFFSET BUFFER    ;point to buffer
        INT   21H
        RET
READ    ENDP
;-------------------------------------------------------------
;write to target file
WRITE   PROC  NEAR
        CALL  STDIR               ;set target dir
        MOV   AH,42H              ;reset file pointer
        MOV   AL,0                ;from start of file
        MOV   CX,POINTTH          ;point target high
        MOV   DX,POINTTL          ;point target low
        MOV   BX,THANDLE          ;get target handle
        INT   21H
        MOV   AH,40H              ;write to file
        MOV   BX,THANDLE
        MOV   CX,BSIZE            ;No. of bytes
        MOV   DX,OFFSET BUFFER    ;point to buffer
        INT   21H
        RET
WRITE   ENDP
;-------------------------------------------------------------
CSEG    ENDS
;*************************************************************
        END   START
```

Introducing SCOPY

The assembly code for SCOPY.EXE, the utility capable of selectively copying files from one directory to another, is provided in Figure 2. When requested, SCOPY reads individual files from a source directory into a buffer area and then writes the file, with the same name, to a designated target directory. SCOPY is organized to:

1. Save the original directory.
2. Clear the screen and position cursor in the upper left corner.
3. Wait for input of source and target directories.
4. Set up the required ASCIIZ strings.
5. Search the source directory and display first match.
6. Offer the option to copy the file to target directory and search next match <Y>, search next match <N>, or quit the program <Q>.
7. Restore original directory.

The search functions (4EH/4FH) assume you have previously invoked function 1AH to declare a 43 byte disk transfer area (DTA). Whenever a match is found, the DTA is filled with information regarding the file's name, date, time, size, and attribute (see Table 2). The last 13 locations of the DTA (bytes 30-42) store the file's name/extension in the form of an ASCIIZ string. It is this portion of the DTA that is displayed after every match. An unsuccessful search terminates the program.

The maximum number of bytes permitted for any read/write operation is 64K. If a file exceeds this limit, it must be duplicated in sections. The question is how many bytes to declare for the read/write buffer? Although the entire ES segment (64K) could be allocated for this purpose, most files are less than 64K and there is no need to waste so much memory. After experimenting with different buffer sizes, I selected 32K as the best compromise between speed and compactness. For the duplication of an 80K file, three read/write passes of 32K, 32K, and 16K are required.

To selectively copy files from the root directory of drive A to A:\TARGET, enter the parameters shown:

```
A>SCOPY

ENTER SOURCE DIRECTORY:
```

```
(EXAMPLES:  A:\, B:\PATH\, C:\PATH1\PATH2\)

A:\

ENTER TARGET DIRECTORY:
(EXAMPLES:  A:\, B:\PATH\, C:\PATH1\PATH2\)

A:\TARGET\

SOURCE
COPY FILE?  <Y> YES    <N> NO    <Q> QUIT
```

If you select the option to copy file <Y>, one of four messages are displayed:

```
FILE COPIED
FILE COULD NOT BE COPIED
NO FILES LOCATED
DISK FULL
```

SCOPY is an excellent utility for duplicating isolated files. Not only is it quick, it eliminates tedious typing. So put SCOPY to work and discover how simple it can be to selectively copy files from one directory to another. •

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these 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 trademarks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosh, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder ii, Dos Disk; Plu*Perfect Systems. Clipper, Nantucket; Nantucket, Inc. dBase, dBASE II, dBASE III, dBASE III Plus, dBASE IV; Ashton-Tate, Inc. MBASIC, MS-DOS, Windows, Word; MicroSoft. WordStar; MicroPro International. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C, Paradox; Borland International. HD64180; Hitachi America, Ltd. SB180; 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

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.**

Technology Resources

K-OS ONE—Single user generic 68000 operating system for your 68000 hardware. It uses the MS-DOS disk format, and includes the operating system with source code (written in HTPL), an editor, assembler, and HTPL compiler. A sample BIOS code and a boot loader are included. This is **not** ready-to-run—you have to install the BIOS on your system, but the source code and language compiler are included **$50**

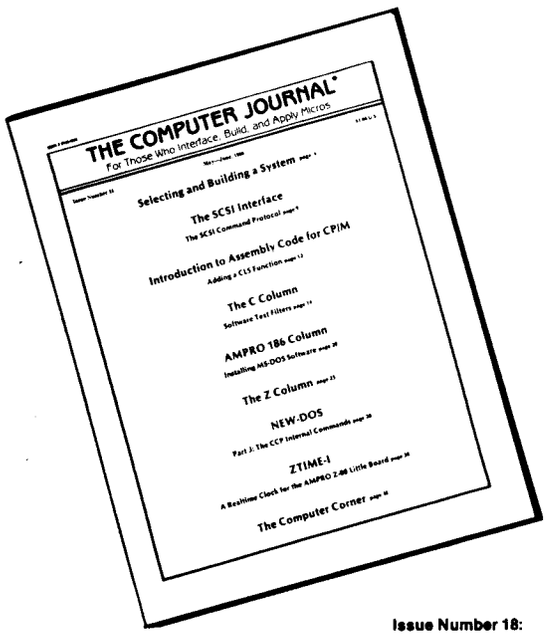
HT-Forth—A full featured, interactive Forth that works with the K-OS ONE operating system. It uses a full 32 bit stack and 32 bit arithmetic to take full advantage of the 68000. Programs are position independent and are limited in size only by the memory available. Source code compiles to inline macros, JSR, or BSR so there is no inner interpreter overhead. Standard ASCII files are used. Includes full screen editor and a Forth style 68000 assembler **$100**

68000Cross Assembler—Written entirely in 8086 assembly language, it is small and fast. All input and output is done with standard MS-DOS calls so it will run on any MS-DOS system, even those which are not totally PC compatible. All 68000 and 68010 instructions are supported. It has conditional assembly, the symbol table is in alphabetical order, and cross referencing is included. Include files are supported so it is easy to assemble big programs, but edit them in small pieces. An equate file can be produced for PROM based programming **$50**

ORDER FROM

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

Visa and Mastercard accepted
Prices postpaid in the U.S. and Canada



THE COMPUTER JOURNAL

Back Issues

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 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 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

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

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

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

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create 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 L.B.
- 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

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

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

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro 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 ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- 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.

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

Issue Number 34:

- Developing a File Encryption System: Scramble data with your customized encryption/password system.
- DataBase: A continuation of the database primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking system.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8086 software to produce modifiable assem. souce code.
- Real Computing: The National Semiconductor NS32032 is an attractive alternative to the Intel and Motorola CPUs.
- S-100 Eprom Burner: a project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System: Part 1-selecting your assembler, linker, and debugger.
- ZCPR3 Corner: How shells work, cracking code, and remaking WordStar 4.0.

Issue Number 36:

- Information Engineering: Introduction
- Modula-2: A list of reference books
- Temperature Measurement & Control: Agricultural computer application
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE!
- Real Computing: NS32032 hardware for experimenter, CPU's in series, software options
- SPRINT: A review
- ZCPR3's Named Shell Variables
- REL-Style Assembly Language for CP/M & Z-Systems, part 2
- Advanced CP/M: Environmental programming

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER
- Information Engineering: Basic Concepts; fields, field definition, client worksheets
- Shells: Using ZCPR3 named shell variables to store date variables
- Resident Programs: A detailed look at TSRs & how they can lead to chaos
- Advanced CP/M: Raw and cooked console I/O
- Real Computing: NS320XX floating point, memory management, coprocessor boards, & the free operating system
- ZSDOS-Anatomy of an Operating System: Part 1

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS--Anatomy of an Operating System, Part 2.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:

- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0—The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

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 _____

#41

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

assembly or "C". That is the big question I always get. Our clients asked the other day if I was doing this project in "C". After I stopped laughing, I explained it was all done in assembly (after all I only have 4K of ROM to use).

Talking of programming, we have had many changes, and all at the last minute. We finished the proto-type and shipped to the client. They started testing and changing their mind. I had chosen the variables and procedures based on a reverse engineering of a previous product. The new product was to do the same thing, but they now want it to do it differently. Our mistake was not producing a software specification of the product. By that I mean, putting in writing just what the software will and will not do. That way when they change their minds (as they have several times) you can charge them for it.

As I look back at the code, I nozticed it turned out much as if it were written in Forth. I feel that is why I like Forth, it is much the same way I normally program. The most efficient way to write code is in modules. I wrote all the I/O units first. Then wrote master modules that just called all the needed smaller units. What I call the main section just calls the master modules in proper order. The original program was straight line, in that one event happened after another was completed. That was my first version of the main section.

Our client started demanding the status indicators be active at any time (multitasking). They want the ready light to be on or off depending on levels of pressure. This means it must be checked continuously no mater what is happening. That of course is not possible in straight line, except through complex interrupt routines. I have chosen not to have the timer be interrupt driven and so changed my main line code into loops. My main now consists of calling subroutines that check pressure, flags, I/O, and time. Based on status checks each of the subs may call other subroutines to do something (indicate Ready) if needed. The thing that was important was being able to rewrite two pages of code in one day to change from straight line to a multitasking like operation. That would not have been possible if I had not modularized the code when I started. I also put all variables or triggering values in the beginning equates. So as our client changed his mind, I only changed one equate and it was done.

As one can most likely guess our budget is running over. We had agreed to design and build 20 units. Looking back I can see now that the designing and prototyping is actually what our boss bid on. The manufacturing of the 20 units has taken considerable time, so much in fact that I feel it should have been a separate bid and contract structure. Proto-typing a product is mostly a time based operation. Buying products for a prototype can come from anywhere, even used or junk boxes. When it comes to making them on a regular bases, you have to deal with suppliers, changing prices, mis-shipped items, incorrect design information, variations in product runs, and enough other problems to keep one or two people working full time. We are spending at least 8 hours (if we are lucky) a week (every week) on coordinating items for the soon to be built 20 units. I am still concerned about some other items like shipping boxes, pre-shipment testing, burn in space, and storage.

We are a small group of design consultants and are using a 100 square foot room for storage and laboratory facilities. When we really start testing the 20 units our hundred feet is going to be too small many times over. We are going to find our needs will be 10 times our current space and no budget to make any changes. I have been down the startup road before with other companies, and it is interesting to see how little it takes to run into the same old problems.

ATs

I decided my home system needed to be at least an AT class machine. I have plenty of systems, but do so much work at home using programs like Orcad that work better with AT style machines. My XT was doing find, but just too slow for CAD and mouse based programs. I purchased an AT clone board from a discount warehouse. It worked fine for the first week or two and then I started getting errors and crashes. My freeze mist said it was a 82C101 chip. The AT clones are mostly custom chip sets and one of them was running very hot (besides the 80286 being VERY hot). I tried more cooling but this was definitely a failed item. I shipped it back and got a new chip installed by them. It works now, but I plan on a few changes.

In one of our newsletters from a local club, the author explained how his system had become flaky. He spent $200+ for a vertical case and more cooling and his problems went away. I started checking the cooling in my XT case, now with an AT board. My question is what cooling. The fan on the power supply only cools the power supply. If you put anything other than a slow running XT in the box, extra cooling is a must. The extra heat from a 80286 which runs very hot normally, a PAL which also runs hot normally, and hard disk, must go some place—preferably outside the unit.

What I am going to try is mounting a fan under the hard drive for cooling it's chips. A portion of the air will travel along under the drives mounting bracket and over the PAL and 80286. I have some cooling fins for the bigger chips, but the 80286 is smaller than the those custom blocks. That means sawing one of the heat sinks down to size and sticking it in place. I plan on just using heat sink compound and not the thermal epoxies. Another option is heat sink compound and two small dabs of silicon hi-temp sealant to hold the sink in place. That would make it possible to remove them later. If this doesn't work at keeping things cool, I may have to use more than one fan. These are 12 Volt fans and I can just stick them on one of the unused (or in parallel) disk drive power lines.

Now What?

I think I have about said it all for now. I have to start preparing for teaching a Junior College class (Introduction to Electronics). I have been trying to find an opening to do a little part-time teaching and one finally came my way. This is looking like a fun class, as all we do is get people into understanding the terms and concepts.

It occurred to me today that I have not mentioned a magazine I get. When we started looking for items for our project a friend suggested *Sensors Magazine.* I have received several issues and find it very useful. It is not thick, but it does cover a few ideas and concepts about using sensors every other month. The list of manufacturers that advertise could be invaluable. They also have a special book that lists all the sensors and their manufacturers. We haven't bought it but suspect it would be worth the money if you are buying lots of different types of sensors. The magazine is a freebee if your business uses sensors.

I still need to hear about having a contest using little CPUs and or articles about the same. I have since found a Forth programmer who is also doing a 68705 project. He also gave me a copy of a Forth for the 6805 series. What I need now is free time to modify the code for the type of Forth I want for my projects. In fact that is all I need now—TIME ..... •

SENSORS
174 Concord St.
POB 874
Peterborough, NH 03458-0874
(603)924-9631

The Computer Corner

by Bill Kibler

Busy, busy, busy....it seems never to stop! Got lots to talk about and little time. Been so busy lately, I have to turn this out in one day. Let's see if I can cover everything.

CADS

Got the version II of OrcadPCB. They have fixed some of the problems. It is now possible to move and place text without the system locking up on you. I was touching up the boards from the project, getting them ready for final production, and had to rearrange the silk screen text. My boss wants the text perfect and all of equal size, so almost all of it had to be changed (orcadPCBI text and pads are a mess). Half way through one editing period (between saves) it locked up and went away (version I that is).

I had real concerns about changing versions before the boards had been completed, but I didn't have any choice. What I needed to do the most was text and pad size changes, just what version one was blowing up on. So out came version two and their convert utility. The BAT file was set up for their special directory arrangement (I restored from floppy) so I had to do it manually. Even doing things differently it all was up and running in 10 minutes. I guess my fears were not founded in reality (this time). Version two has a few bugs, but for the most part works OK.

My biggest complaint was being unable to use a printer with the program. No way to proof your work is not acceptable for me. They have changed this, although when I ran the printer program it printed the drill locations OK, but produced a blank silk screen board. I need to check this out more (may not have a correct switch set in their software). I still think I will use FPLOT for the board layouts as it is about five times faster than the printer routines. The Orcad printer routines in both SDT and PCB are very slow, although not as slow as a plotter.

We outputted the files to a 10 inch per second plotter and it took a half hour to plot one side of the board. When you are proofing the board what you want is speed and usable detail. That is why I still think FPLOT is best by far. The text handling is much better than it was, however you still can not edit the actual text. What you can do, and it does work, is delete the text and then write the new or corrected text back. I do not really find this acceptable, but maybe the next upgrade will finally get it (and other problems) right. I still like their support as we received their update without having to do anything (as long as you are still in warranty—1 year from date of purchase).

My writing about CAD problems prompted one group to send me their CAD program. I talked to one of the people on the phone and he indicated it was a pretty good program with lots of features that Orcad didn't have. When I asked him about how easy it was to use, he indicated that most people had to talk to him to get started. Well, I received their package and was ready to send it back after looking at the documents. Not feeling like that was much of a review I will load it on the hard disk and give it a try. The reason I wanted to send it back was it is very similar to ACAD. Personally I dislike ACAD a real big bunch. To me ACAD is a perfect example of how not to produce a user interface. It is true that these products are older designs and a lot of knowledge has developed over user interfaces since they came out.

Without having used the program, this other CAD product is scoring a perfect zero. I see lots of programs and I must say that a large percentage of them would get this mark. Why? I score documentation as being almost as important as the actual program. Large slick or good looking books is not what I am after. They must be well organized, reflect the design philosophy of the software, have plenty of indexes, a tutorial section (step by step instruction with sample screens), and have many levels of user ability supported. Most older styles of manuals have a section in which all the commands are listed alphabetically. I find that almost useless. When help is needed, it is usually based on some command activity in which we have no idea what the command is, just the function we want.

Let me say that some of the biggest software companies still don't have it right either. I prefer Wordstar over Wordperfect because WS has the menus broken down into understandable groups (file, screen, misc.), whereas WP just has an alphabetized list of commands. Another example is when Turbo Pascal first came out in the CP/M days, they had two manuals. The first manual was the programmers guide which dryly explained commands. The other was a tutorial approach to using TP. I know of few people who used the programmers guide, but most used the tutorial as it provided better and less technical advice on using commands. The examples in the tutorial also helped clarify many unsaid (or unwritten) concepts.

Simply put I feel Orcad is doing well because it is a current product using the more recent concepts in software engineering. Those are: a good menu driven system (5 or 6 main menus), easy switching between mouse and keyboard input, tutorial and topic based documentation, a low level of technical-ese used in the documentation, and a common sense use of commands. The way to test for this is by seeing how much studying it takes before you can use the program. I was using Orcad SDT after 20 minutes of minor instruction by my boss (no manuals needed to start). As I continued to use it, I was able to increase my skills and abilities without extra or special courses on the program (just browsing the book). Both Wordperfect and ACAD fail those tests.

68705

Well it is great to see other magazines reading my columns, or maybe that Motorola's learning package is turning people on to their devices. MicroCorucopia has several articles on the education package (seemed like edited down version of Motorola's work to me) plus several short articles on using the I/O. I am glad to see this, as a lot of projects now done on bigger CPUs are actually better situated for the little guys. It seems like so much more fun too!

I have gotten a few comments on a contest, and in fact one entry. Using a 1802 to blink lights by Lee Hart. Seems he had manufactured an 1802 based system and had built this some time back to show how their system worked. He also explained to me his two good features of the 1802—low current needs and a Forth ROM operating system. He wrote the Forth ROM system and I have asked him to write us an article about it. What I want him to cover is how and why Forth made programming in real situations better than