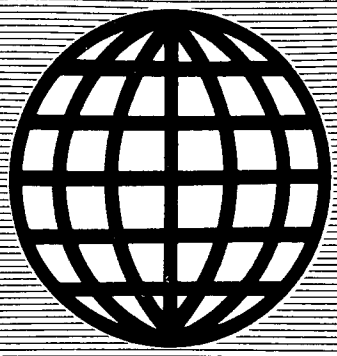

The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 35 November / December 1988 $3.00

All This and Modula-2
Is This the Next System Programming Language?

A Short Course in Source Code Generation
Disassembling 8086 Code

Real Computing
The National Semiconductor NS32032

S-100 EPROM Burner
Building the Digital Research Board

Advanced CP/M
ZSDOS and File Systems

REL-Style Assembly Language for CP/M and Z-System
Part 1: Choose Your Weapons

ZCPR3 Corner
Shells and Patching WordStar 4.0

ISSN # 0748-9331

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

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Circulation
Donna Carlson

Contributing Editors
Joe Bartel
C. Thomas Hilton
Bill Kibler
Bridger Mitchell
Bruce Morgan
Richard Rodman
Jay Sage
Barry Workman

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

Send subscriptions, renewals, or address changes to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, Montana, 59912.

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

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

The COMPUTER JOURNAL

Features

Issue Number 35

November / December 1988

Columns

you see here.

THE AMPRO LITTLE BOARD™/286

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

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

Compare features.
Both systems offer:

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

But only Little Board/286 offers:

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

*AT is a Registered Trademark of IBM Corp.

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

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

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

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

Editor's Page

Where There's Smoke, There's Fire

Any one who listens to the national news must be aware that the Northwestern portion of the United States is burning up. We have been casually following the fires in Yellowstone National Park, which is an area that we have visited. Now, there is a very serious fire in Glacier National Park, which is only about 25 miles away from us.

We are following the Glacier Park fire (named the Red Bench Fire) very closely because it is close to home, and it is affecting places and people we know. We usually take most of our day trips before Memorial Day or after Labor Day in order to avoid the tourists. We had planned a trip up the Red Meadow Creek drainage this week—and that's where the fire started! Now, that area is closed. In fact the whole state is closed to recreational activity until we get adequate moisture.

In case you've wondered why we aren't always available to answer the phone, we often work early in the morning and late in the evening so that we can spend some time out in the mountains during the middle of the day. Keep calling till you get us.

de facto Standards

Most of the computer industry standards which directly affect our day-to-day operations are determined by the market instead of by formal committees. The market can only accept or reject what is offered to it, and we don't always agree with the choices it makes, but there are times when we have to go along with it and to follow the market's lead. It is one thing to follow our own interests where we can (and do) work with whatever we choose, as long as we fully recognize that this may not be of any interest or use to anyone else—at least it may not be of any commercial interest. But, if we want to benefit financially from the fruits of our labors, we have to either perform our work-for-profit in commercially accepted areas, or else to be a pioneer and convince the market to follow. And, I'm sure that you have heard the comment, "You can always recognize pioneers by the arrows in their back."

We have to differentiate between what we do by matter of choice and what we do because of economic necessity, and to clearly separate the two. It would be much simpler if our personal and commercial interests were in entirely separate fields—but then we wouldn't be working on something which we enjoy.

One of the current de facto standards is the 5.25" MS-DOS 360K disk format—which is becoming the standard format for information exchange, regardless of the operating system. We all realize that our home-brewed 32032 or 68030 system will not run Lotus 1-2-3® just because it can read the MS-DOS disk format, but our systems should be able to read and write data files on an MS-DOS disk format. Joe Bartel used the MS-DOS format for his Hawthorne Technology 68000 board, and Richard Rodman is using it for the NS32 system. Every system using a 5.25" floppy should either use the MS-DOS 360K as its native format, or at least should be able to read and write to that format. The AMPRO Z80 Little Board® comes with a utility to transfer programs between CP/M and MS-DOS disks, but it does not allow editing or other operations directly on the MS-DOS disk. Bridger's DosDisk is a great improvement, because it allows programs running on a Z80 CP system to use files stored on a 360K MS-DOS disk directly without copying.

There does not seem to be much activity with the AT 5.25" 1.2 Meg disk format. It requires special high density disks, 360K formats written in a 1.2 Meg drive can not be read reliably in a standard 360k dive, and I have heard a lot of complaints about read failures with the 1.2 Meg disks in a 1.2 Meg drive. It appears that this format may end up being an orphan.

The MS-DOS 3.5" 720K format will very likely become the de facto data interchange format for that size disk. There have been reports of problems with the 3.5" 1.44 Meg, and this may end up being another orphan.

Another standard is the dBASE III® file format. There are more and more programs becoming available which can access the files directly without having to first export the data to an ASCII or other interchange format. Some programs can even create and manipulate the files without ever using dBASE.

One of the improvements in WordStar Professional Version 5® is the ability to merge information from dBASE files into a document. Version 5 also includes their MailList utility for creating and working with simple databases, but I haven't had a chance to check its data file structure.

1st Class Expert Systems Fusion® program accesses and manipulates dBASE files, and provides the opportunity for establishing expert system shells for databases. These expert system interfaces will be very important in facilitating database access for end users. We are working on a full evaluation of Fusion.

"What changes will we see by the end of 1990?"

dBC III® from Lattice provides a C library with functions to perform most of the basic database operations using dBASE III compatible files. It can either create the files for its own use, use dBASE files provided by someone else, or create files to be used with dBASE. The functions are intended to be linked into your C program. You have to do a lot of programming, but you have full control over your program. Using the powerful resources available to a C programmer enables you to accomplish tasks which are very difficult to program in dBASE—and you, or someone else, can still use dBASE III with the files. You can distribute the compiled EXE programs without any

(Continued on page 36)

All This and Modula-2

Pascal with Scope and Parameter Passing

by Dave Moore, Cerenkof Computing, Australia

Modula-2 was designed by Niklaus Wirth (pronounced Veert), the same man who designed Pascal. However, Modula-2 is not just "Pascal-Over-Extended"; it is a new language in its own right.

When Niklaus Wirth designed Pascal, he wanted to produce a language which was suitable for beginning students in computing. Such a language had to be simple to learn but had to also be able to support the sophisticated algorithms the students would study in later courses.

Professor Wirth had been a part of the effort to define Algol-68, which is the successor to Algol-60. He had become disillusioned with the complexity of this language. As a result, he wanted to show that a simple language could also be a powerful language.

The motivation for Modula-2 is quite different; Modula-2 is intended to be a systems programming language. When the language was designed, the professor and his colleagues were also designing a computer which we now know as the "Lilith". All the software for the Lilith, including the operating system, is written in Modula-2.

An operating system is a large program. It would be impossible to write such a program as one long piece of code, as is required by standard Pascal. It would take forever to edit and to compile, and only one person would be able to work on it at a time: one hundred man years effort would take one hundred years to complete!

A better approach was needed. It must be possible to divide the task up into modules so that several modules can be written by different programmers in parallel.

Also, it must be possible for each programmer to specify exactly what he is writing—and how to call it—even before he writes the code. Otherwise, you could not write a module until all the modules it calls were completed.

Modula-2 allows just this. Programs are divided into *modules*. Each Module has two parts, a *Definition Module* and an *Implementation Module*. The definition module tells the outside world (of the program) what the module does, and how to access it. The implementation module actually does the work.

Suppose that you and your friends are working together to write a program. The first thing you would do is meet together to break the program into logical pieces, then assign each programmer one or more of those pieces.

For example, let's suppose that you are writing a flight simulator. One obvious way to break the program is:

```
Controls Module                          (Bill)
Speed, Position, and Attitude Module     (Les)
Instrument Display Module                (You)
Scenery Display Module                   (Linda)
```

You have to write an instrument Display module. But the values you need to display, (airspeed, altitude, etc.), are all part of the Position, Speed and Attitude module. Before Les leaves the room, he must write a definition module for his module. It could

---LISTING ONE---

```
MODULE CompDir;

(* CompDir - Compare two directories

 Dave Moore
 Cerenkof Computing
 PO Box 305, Wynnum Central 4178  AUSTRALIA

 Call is COMPDIR  wild card,wild card,list file [options
  Where each wild card produces one of the lists to compare
  For example, to compare two discs, do: COMPDIR a:*.* b:*.*
  Options are any of the following:
  T compare only file type (by default, the whole name is
compared)
   (by file type, I mean the (3) characters after the dot)

  N compare only file name
  C if file names match, check file contents as well
   These two options are mutually exclusive

   E Output file names which match to list file
   A Output file names which have been added to the second
list
  D Output file names which have been deleted from the
first list
  M Output files which have been modified to the list file
   These options may be combined

 The program can be very useful for updating discs based on
their existing contents, which is an area in which sweep is
weak.
 This is a modified and enhanced version of the original
CompDir program which was written for expository purposes.
The program is written in FTL Modula-2.
 *)

FROM  SYSTEM   IMPORT ADDRESS,ADR,TSIZE;
FROM  Command  IMPORT Parameter,Parclass,GetParams;
FROM  Terminal IMPORT WriteString,WriteLn,Write;
FROM  Sort     IMPORT SortRecords;
FROM Files     IMPORT FileName,FILE,Create;
FROM GetFiles IMPORT GetNames;
FROM Strings  IMPORT Length;
FROM Streams  IMPORT
STREAM,Connect,Disconnect,WriteChar,Direction;
IMPORT InOut;
FROM FileOps  IMPORT FileComp;
CONST MaxFiles=512; (*max files to be matched*)

TYPE
 CompClass=(Whole,Name,Type);
 FileRecord=RECORD
  Device:CHAR;
  Name:ARRAY[1..8] OF CHAR;
  Type:ARRAY[1..3] OF CHAR;
  END;
```

```
 PFileName=POINTER TO FileRecord;
 FileArray=ARRAY[1..MaxFiles] OF FileRecord;
VAR
 Param:ARRAY[1..4] OF Parameter;  (*area for command line*)
 Count:INTEGER;    (*parameters*)

 LeftNames,RightNames:FileArray;
 LeftNameCount,RightNameCount:INTEGER;

 Compclass:CompClass=Whole;
 OutputAdded:BOOLEAN=FALSE;
 OutputEqual:BOOLEAN=FALSE;
 CompareContents:BOOLEAN=FALSE;
 OutputDeleted:BOOLEAN=FALSE;
 OutputModified:BOOLEAN=FALSE;

 ListOpen:BOOLEAN=FALSE;
 List:STREAM;
 ListF:FILE;
PROCEDURE CrackFileName(Text:ARRAY OF CHAR;VAR
Rec:FileRecord);

 (* Convert the text form of a file name to the
  expanded form*)
 VAR i,j:CARDINAL;
 BEGIN
  WITH Rec DO
   IF Text[1]=':' THEN
    Device:=CAP(Text[Ø]);
    i:=2;
   ELSE
    Device:=' ';
    i:=Ø;
    END;
   j:=1;

   (* pick up file name *)

   WHILE   (i<=HIGH(Text))
       AND (Text[i]<>Øx)
       AND (Text[i]<>' ')
       AND (Text[i]<>'.') DO
    IF  j<=HIGH(Name) THEN Name[j]:=Text[i] END;
    INC(j);
    INC(i);
    END;
   FOR j:=j TO HIGH(Name) DO Name[j]:=Øx END;

   (*Note that I use zero instead of blank fill to simplify
listing of the names*)
   j:=1;
   IF Text[i]='.' THEN INC(i) END;
   WHILE   (i<=HIGH(Text))
       AND (Text[i]<>Øx)
       AND (Text[i]<>' ') DO
    IF  j<=HIGH(Type) THEN Type[j]:=Text[i] END;
    INC(j);
    INC(i);
    END;
   FOR j:=j TO HIGH(Type) DO Type[j]:=Øx END;
   END;(*WITH*)
  END CrackFileName;

 PROCEDURE Usage;
 BEGIN
  WriteLn;
  WriteString('Usage is:   COMPDIR wild card file name');
  WriteString(', wild card file name [, list file]
[/options]');
  WriteLn;WriteLn;
  WriteString(' Options are:  ');WriteLn;
  WriteString('  T compare only file type ');WriteLn;
  WriteString('  N compare only file name');WriteLn;
  WriteString('  C compare contents of matching
files');WriteLn;
  WriteString('  E Output file names which match to list
file');WriteLn;
```

look something like this:

```
DEFINITION MODULE SPAM;
 (*   Speed Position Attitude Module   *)
  VAR  Altitude,Northing,Easting:INTEGER; (*in meters*)
      Bank, Direction,DescentAngle:CARDINAL; (* radians*1024 *)
      Speed:CARDINAL;                    (* meters/second *)
 END SPAM.
```

This definition module contains a list of all the elements of Les's modules which you can "see" from your module. There will be other objects which you cannot see. For example, when Les gets home and starts thinking about how to write his part of the program, he will decide he needs a variable which gives the angle of attack of the wing, so he can calculate the amount of lift.

Because this variable ('AnlgleofAttack' say), is not in the definition module, it is not "visible" to other modules. If later during testing, the variable is found to contain the wrong value, Les knows he only has to examine his code for the error: no-one else in the group can change the variable.

Just as importantly, if he wants to change the way the value is represented, (such as changing the units used), he knows that it cannot affect the operation of any other module.

When you write your module, you will IMPORT the items you want to use. Your module might start like this:

```
MODULE Instruments;
FROM SPAM IMPORT Altitude,Bank,DescentAngle,
     Direction,Speed;
```

These are the items that you want to use. You have not imported the Easting and Northing variables because you do not need to display the plane's current position on the instruments. Notice that Linda will need those variables to determine what can be seen through the window. She will import those variables, but not some of the others.

You do not need to redefine your imported objects. By giving the name of the module and the name of the required object, its definition is retrieved automatically.

The fact that the original definition is accessed by your module means that you cannot inadvertently give a writing declaration. In other languages, this problem can be reduced by using "include" files. This does not prevent your from changing an include file and forgetting to recompile all the modules which use it.

With Modula-2, when you link your modules together to form a completed program, the Modula-2 linker will check that all the modules are up to date. This prevents your from accidentally linking in a module which was compiled with an old version of a definition module—one that may not agree with the current definition of the module.

Standard Modules

Every Modula-2 compiler comes with a set of standard modules to perform useful functions. At the very least, you will receive modules to perform file and terminal input-output and storage management. Most compilers will also come with other predefined modules. (Note, FTL Modula-2 includes all defined standard modules and includes the sources to those files.)

In a good compiler, you will receive these modules in source code, so that you can modify them for your own requirements and use them to develop other modules.

These modules make it very easy to write quite sophisticated programs. Let us look at an example.

I often want to compare two disks to determine what files they have in common. For example, if you are making a disk to send to a friend (or a customer), it is very useful to be able to check that no files have been left off the distribution disk.

Program 1 is a Modula-2 program to do this. It reads the directories of both disks, sorts the file names into alphabetical order, and prints the file names in two columns with matching file names

printed next to each other. For example:

```
A> CompDir a:*.mod b:*.mod
CAT.MOD
                        COMPDIR.MOD
LIST.MOD                LIST.MOD
XERA.MOD
A>
```

The program takes the form of a module. Because the module does not have any symbols which can be used by other modules, it is not divided into separate definition and implementation parts. Unlike most other languages, Modula-2 does not have a unique "main Program" part. Rather, every module may contain statements which are to be executed when the program is loaded.

The only rule is that, if module A imports module B, then module B will be initialized first. Thus, in the case of our program, we know that all the modules it imports are ready to go when its "main line" executes. At the beginning of the program, there is a list of IMPORT statements. For example:

```
FROM Command IMPORT Parameter,Parclass,GetParams;
```

"Command" is the name of a standard module which comes with the FTL Compiler. It interprets the command line used to call the program. "Parameter" and "Parclass" are types while "GetParams" is the procedure which is called to perform the parsing.

These IMPORT statements are followed by a number of declarations which look very much like declarations in Pascal. Notice that the imported types can be used in exactly the same way that they would be used if defined locally.

Of the procedure definitions which come next, two are worth close examination. The first procedure, "Compar", is a key comparison routine for the supplied sort module. Line 65 of the program contains a call to the sort procedure in the sort module. "Compar" is one of the parameters in this procedure call. By writing a short key comparison procedure, the sort module can be used to sort any type of data in any collating sequence.

The declaration for this module must exactly match that required by the procedure in the sort module. The parameter must be of type:

```
TYPE     KeyProc=PROCEDURE (ADDRESS,ADDRESS):BOOLEAN;
```

In Modula-2, procedures have types, and can even be used as the types of variables!

ADDRESS is a standard Modula-2 type which is compatible with any pointer type. Because the key comparison routine needs to compare file names, the parameters are changed to have the type PFileName (POINTER TO FileName) before the comparison is performed. This is an example of the "type breaking" facilities in Modula-2: you can tell the compiler to treat a variable as a different type for the purposes of an expression.

The second interesting procedure is ExpandNames. This procedure calls GetNames to return a list of file names which match a wildcard parameter, removes any leading drive designator and then calls SortRecords to sort the filenames into order.

Finally, the main program, at lines 97 through 147, calls GetParams to parse the command line, call ExpandNames to expand each of the wildcard filenames into a list of matching file names, and then uses a classical merge algorithm to compare the two files and print the lists side-by-side.

This program took me no more than two hours to write, because many of the elements I required already existed and could be easily accessed using the module structure of Modula-2. In another language, I would either have spent several hours reinventing the wheel or else searching for old programs that used routines like the ones I needed now. What is more, because of the strong typing, once a clean compile had been obtained, the program ran at the second attempt.

```
  WriteString('  A Output file names which have been added
to the second list');WriteLn;
  WriteString('  D Output file names which have been deleted
from the first list');WriteLn;
  WriteString('  M Output file names of modified files
');WriteLn;
  WriteString('   (Implies C option)');WriteLn;
  HALT
  END Usage;

 PROCEDURE GetOptions(p:Parameter);
 VAR i:CARDINAL;
 BEGIN
  WITH p^ DO
   i:=0;
   LOOP
    CASE Chars[i] OF
     0x:EXIT
     'T':Compclass:=Type
     'N':Compclass:=Name
     'E':OutputEqual:=TRUE
     'A':OutputAdded:=TRUE
     'D':OutputDeleted:=TRUE
     'M':OutputModified:=TRUE;
         CompareContents:=TRUE
     'C':CompareContents:=TRUE
     ELSE Usage;
     END;(*Usage*)
    INC(i);
    END;
   END;(*WITH*)
  END GetOptions;

 PROCEDURE Compar(a,b:ADDRESS):BOOLEAN;

 (* key comparison routine for the sort module
  Returns TRUE if a is greater than b

  This routine always compares the full name as it is used
  for sorting. The equality routine only compares the part
  of the name requested by the flags. As a result, a test
  for equality should be made first*)

 VAR p,q:PFileName;
 BEGIN
  p:=PFileName(a);
  q:=PFileName(b);
  CASE Compclass OF
   Type:IF  p^.Type<>q^.Type THEN RETURN p^.Type>q^.Type
        ELSE RETURN p^.Name>q^.Name END;

   Name,Whole:IF p^.Name<>q^.Name THEN RETURN
p^.Name>q^.Name
         ELSE RETURN p^.Type>q^.Type END;
   END;(*CASE*)

  END Compar;
 PROCEDURE Equal(p,q:FileRecord):BOOLEAN;

 (* compare to names for equality
  patterned after the preceding routine*)

 BEGIN
  CASE Compclass OF
   Type:RETURN p.Type=q.Type
   Name:RETURN p.Name=q.Name
   Whole:IF p.Name<>q.Name THEN RETURN FALSE
         ELSE RETURN p.Type=q.Type END;
   END;(*CASE*)

  END Equal;
```

```
PROCEDURE ExpandNames(Param:Parameter;VAR Names:FileArray;
   VAR NameCount:INTEGER);

(* Expand wild card file name into list of files
 and sort result *)
VAR i,j:INTEGER;
 NameText:ARRAY [1..MaxFiles] OF FileName;
 Text:ARRAY[0..5] OF CHAR;
BEGIN
 IF (Param^.Chars[2]=0x) AND (Param^.Chars[1]=':') THEN
  Text:=' :*.*';
  Text[0]:=Param^.Chars[0];
  GetNames(Text,NameText,NameCount);
 ELSE
  GetNames(Param^.Chars,NameText,NameCount);
  END;

 (*crack names*)

 FOR i:=1 TO NameCount DO
  CrackFileName(NameText[i],Names[i]);
  END;

 IF NameCount>1 THEN
  SortRecords(ADR(Names),CARDINAL(NameCount),
   TSIZE(FileRecord),Compar);
  END;

 END ExpandNames;

VAR i,j:INTEGER;
PROCEDURE WriteToList(VAR N:FileRecord);
BEGIN
 WITH N DO
  InOut.WriteString(Name);
  WriteChar(List,'.');
  InOut.WriteString(Type);
  InOut.WriteLn;
  END;
 END WriteToList;

PROCEDURE WriteName(VAR N:FileRecord;ToList:BOOLEAN);

(* output a file record as a valid file name
 IF Tolist, output it to list as well*)

VAR i:CARDINAL;
 ch:CHAR;
BEGIN
 WITH N DO
  IF Device<>' ' THEN
   ch:=':'
  ELSE
   ch:=' '
   END;
  Write(Device);
  Write(ch);
  IF ToList THEN
   WriteChar(List,Device);
   WriteChar(List,ch);
   END;
  WriteString(Name);
  Write('.');
  WriteString(Type);
  IF ToList THEN
   WriteToList(N);
   END;

  (*Now pad name out to 16 columns*)

  FOR i:=Length(Name)+Length(Type)+4 TO 16 DO
   Write(' ');
   END;
  END;
 END WriteName;
PROCEDURE PrintLeft;
BEGIN
```

```
 WriteName(LeftNames[i],OutputDeleted);
 WriteLn;
 END PrintLeft;
PROCEDURE PrintRight;
BEGIN
  WriteString('                ');
 WriteName(RightNames[j],OutputAdded);
 WriteLn;
 END PrintRight;

PROCEDURE CompareOK(f1,f2:FileRecord):BOOLEAN;

VAR fn1,fn2:FileName;

PROCEDURE SetName(f:FileRecord;VAR fn:FileName);
VAR i,j:CARDINAL;

PROCEDURE Put(ch:CHAR);
BEGIN
 fn[i]:=ch;
 INC(i); '
 END Put;

BEGIN
 i:=0;
 WITH f DO
  IF Device<>' ' THEN
   Put(Device);
   Put(':');
   END;
  j:=1;
  WHILE (j<=HIGH(Name)) AND (Name[j]<>0x) DO
   Put(Name[j]);
   INC(j);
   END;
  j:=1;
  Put('.');
  WHILE (j<=HIGH(Type)) AND (Type[j]<>0x) DO
   Put(Type[j]);
   INC(j);
   END;
  WHILE i<=HIGH(fn) DO Put(' ') END;
  END;
 END SetName;
BEGIN
 SetName(f1,fn1);
 SetName(f2,fn2);
 RETURN FileComp(fn1,fn2)
 END CompareOK;

VAR reply:INTEGER;
BEGIN


 GetParams(Param,Count);
 IF (Count>0) AND (Param[Count]^.Class=option) THEN
  GetOptions(Param[Count]);
  DEC(Count);
  END;
 IF Count<2 THEN Usage END;
       IF Count=3 THEN
  Create(ListF,FileName(Param[3]^.Chars),reply);
  IF reply<0 THEN
   WriteString(' Could not open output file ');
   WriteString(Param[3]^.Chars);
   WriteLn;
   HALT;
   END;
  Connect(List,ListF,output);
  InOut.SwitchOutStream(List);
  ListOpen:=TRUE;
 ELSE
  IF OutputAdded OR OutputDeleted OR
     OutputEqual OR OutputModified THEN
   WriteString(' A D E and M options require output file
name$')
   WriteLn;
   Usage;
   END;
  END;
 ExpandNames(Param[1],LeftNames,LeftNameCount);
 ExpandNames(Param[2],RightNames,RightNameCount);

 (*now perform merge pass of two lists*)

 i:=1;
 j:=1;
 WHILE (i<=LeftNameCount) AND (j<=RightNameCount) DO
  IF Equal(LeftNames[i],RightNames[j]) THEN

   WriteName(LeftNames[i],OutputEqual);
   WriteName(RightNames[j],FALSE);
   IF CompareContents AND
     NOT CompareOK(LeftNames[i],RightNames[j]) THEN
    WriteString(' Files Differ ');
    IF OutputModified THEN
     WriteToList(LeftNames[i]);
     END;
    END;
   WriteLn;
   INC(i);
   INC(j);

  ELSIF NOT Compar(ADR(LeftNames[i]),ADR(RightNames[j]))
THEN

   PrintLeft;
   INC(i);

  ELSE (*Leftnames[i]>RightNames[j]*)

   PrintRight;
   INC(j);
   END;(*IF*)
  END;(*WHILE*)

 (*process stragglers*)

 WHILE i<=LeftNameCount DO
  PrintLeft;
  INC(i);
  END;

 WHILE j<=RightNameCount DO
  PrintRight;
  INC(j);
  END;
 IF ListOpen THEN
  Disconnect(List,TRUE);
  END;
END CompDir.
```

If you do a lot of programming, you probably have modules to perform sorting, command line parsing, and wildcard expansion. But how long will it take you to find them? When you do find them, how much will you need to change them to meet your new requirements?

With Modula-2, you can build up a library of useful modules, and the definition module construct makes finding routines relatively easy. Also, as demonstrated by the use of the Sort module in the example program, although the language lacks the "generic" facilities of Ada, it is powerful enough to support the writing of re-usable modules in a clean and streamlined way.

Modula-2 is an Ideal language for writing large programs, especially on small machines or when memory space is limited. Even on large machines, many experts argue that Modula-2 as a better language than Ada, Pascal, and C because of its relative simplicity and portability. ■

Editor's Note: This is the beginning of a regular section covering Modula-2, which will include a public domain user's disk library. Your articles, letters, comments, and disk contributions are needed to make it grow!

A Short Course in Source Code Generation

Disassembling 8086 Software

by Clark A. Calkins, C.C. Software

Eitor's Note: A similar article covering source code generation for Z80 systems was published in issue #27.

Introduction

What is source code generation all about? Well, I consider this as the process of creating usable program source code in some language that is equivalent to an initial executable program. By usable I mean that the source code has been sufficiently documented or commented such that you or someone else schooled in the language can understand what is going on.

If you already have an executable program, why would anyone want the source code? The truth is, many users would not. But if you ever wanted to change a program, source code makes it much easier. In some cases, it is not practical to change a program without the source. Besides the necessity or desire to change a program, some people would just like to understand how it works.

Why is it that software producers very seldom make source code available for their products? There are several reasons (or excuses) for not releasing the source code. These include:

1) "...you don't need it, program XYZ works perfectly as it is."

2) "...you could then make copies and give these to others."

3) "...it's too complicated, you couldn't understand it."

and so on. Actually, the real reason is that software writers feel that their programs represent private and creative thoughts and they loose privacy if the source code gets out of their control. There is a mystique to writing programs and programmers tend to keep it that way. A form of "job security."

However, users do have a legitimate requirement for source code. After all, the programmer wrote the program to do what he/she thought was the best. Users generally have different ideas. Program options are set to default values that the programmer found to be most convenient. Users may like to change these. Some programs come with installation instructions that do allow some flexibility in this regard, but this is not always enough. If you ever wanted to learn how to write an operating system or compiler and only had a book or two for reference, you would then see the advantage of having some source code to refer to. This is indispensable!

Can the source code for any program be produced? Theoretically the answer is yes. However, from a practical standpoint, only a few programs would be worth the effort involved. The idea to keep in mind, is that if the computer can execute the program, it can be disassembled. This is because the process of disassembling a program is really a conversion from one language (machine code) to another. Like translating a book from German to English except there are no ambiguities to worry about. Documenting the program now involves just time and effort (maybe quite a lot of these).

The specific type of source code generation I am going to be dealing with here, is the creation of 8086 assembly source code. If it were desired to create Pascal or Fortran source code (assuming the program was written in one of these languages), the first step would be to produce assembly instructions and then convert groups of instructions into source statements. This would require a thorough understanding of the code produced by the compiler. A difficult task indeed, but it can be done.

In addition, I will be assuming that MS-DOS is the operating system being used and that it is desired to disassemble a normal transient program and not a ROM or other type of program storage. These are slightly more complicated as the execution address may differ from the physical address.

The disassembler I will be using for the examples is my own (the Masterful Disassembler or MD86 for short) but most any quality disassembler can be used. I will try to keep this as "generic" as possible, but the figures will come directly from MD86 screen displays.

The choice of a disassembler is important. You want it to do as much of the work as possible. The features I find indispensable are:

- Interactive and visually oriented.
- Ability to add your own names for labels.
- Ability to insert comments.
- Support for many data types.

In addition, a disassembler can be more helpful if it can:

- Recognize the 80x87 and 80x86 instruction sets,
- Mark "questionable" instruction sequences,
- Automatically insert helpful comments,
- Recognize common MS-DOS functions.

If the disassembler you use does not allow label names and comments to be entered, then you will have to work from a printed listing and use an editor to keep track of things. This is a lot more trouble on large disassemblies because up-to-date listings may take hours to generate and you will be tempted to try and get by with older ones. This will sooner or later cause duplication of effort as you re-comment some areas a second time.

The examples I will be using have come from actual programs but the addresses listed are fictitious. These have been chosen to cover a few key areas of disassembling but are by no means exhaustive.

Disassembling programs is more art than science. After you have done one, the next becomes easier. Just like programming.

Disclaimer

The guidelines contained herein are for educational purposes only. The legality of disassembling a program is not totally free from doubt (although it is done on a routine basis). Software licenses may impose limitiations that the user should be awre of. It is certainly not the intent of this article to deprive any software producers of rightfully earned revenues.

Choosing the Proper Candidates for Dissection

Before setting out to produce source code for that super new program, you first want to know if you can succeed. There are some guide lines that I use before I embark on a new project. I ask myself these questions:

- Is there a real use for the source code for this program?
- Is the program small enough that I can complete this project in a reasonable amount of time?
- Was the program written in assembly or at least compiled with a good (non-threaded) compiler?
- Is the program static or does it move itself around in memory?

The answers to these questions, when taken together, help me to decide whether or not to try to disassemble a program. I really try to avoid programs that have been written in a high level language. These can be real messy. The program size is also very important. I have found that, on the average, every processor instruction, uses 2 bytes of memory if it is 8086 code. Data areas seem to average 4 bytes per line. This means that an 8k program would result in 3000 lines of code if it contained about 2k of data tables. A final source file for this may take up 100k of disk space. I don't recommend users try disassembling programs in excess of 16k.

I have disassembled many programs over the years. Some times for fun and other times on a consulting basis. When I am trying to estimate the number of hours required to disassemble a program, I use the following formula.

ManHours = 3.5 * (Program_size_in_kbytes)^1.5

This assumes that the program is written in assembly language and contains 20% data and 80% instructions. This would prove to be overly optomistic on hardware dependent type programs (like a BIOS or disk formatter) and pessimistic on real short programs (< 1k). So far this has proven to be a reasonable starting point. If a 4k program were to take you a couple of weeks, then a 32k program would take a year. And a 128k program over 7 years! Now you can see why the program length is so important!

How can you tell if a program was originally written in assembly language and is this really important? With some of today's optimizing compilers, this question becomes less important. But in general, when I first look at a disassembly of a program I want to see if there is a consistent flow of logic to the code. When I see code that is composed of a seemingly endless series of calls to subroutines grouped at one end of the program (either low or high addresses), I know this came from a threaded type compiler and will be a bear to disassemble. I like to see subroutine calls followed by code that uses the 8086 flag register (like "CALL 1234" followed by "JZ 1267"). This is the way assembly programmers work but few compilers are smart enough to do this. Most would have inserted a "AND AX,AX" or similar instruction. Additionally, compilers produce code that executes under all conceivable conditions. Thus there will be many long or far jump instructions and few short (2 byte) ones. One special note about NOP instructions. Assemblers, particularly MASM, insert NOPs as filler bytes. If these follow a short jump instruction, this would not indicate that a compiler was used. However, if a NOP is the destination of a jump instruction then the odds are good that a compiler produced the code (and not a particularly good one at that). If you find that you cannot tell if the code was produced by a compiler or an assembler, then it probably does not make much difference.

Over the years, users have suggested various projects for disassembly. These have ranged from fairly reasonable ones like MS-DOS, and Turbo C to absurd ones like EDLIN and LOTUS-123. Who even uses EDLIN and who has enough time to disassemble LOTUS? Some of the Norton Utility programs (like SI.EXE) would make reasonable projects.

The Process of Generating Usable Source Code

Prior to disassembling a program you want to learn as much as you can about how it works. You need to know exactly what the program does and have a reasonable idea of how it does it. When you understand what a program does on a gross level, you will be able to understand the finer details when you get to them.

When I start on a disassembly project, I find it is generally a four step process.

- Understand what the program does and how it functions.
- Identify the instruction and data areas. Determine the type of data being stored.
- Isolate the subroutines (identify the lowest level ones first). Label and comment these routines.
- Comment and label the main program code.

The first step of this process occurs before you use the disassembler. You will want to look over as much material (user manuals, reference manuals, etc.) as you can. Initially you will be most interested in locating parts of the program in memory. Hopefully the reference manual contains a memory usage map or words to describe how memory is allocated and what use is made of it.

The three remaining phases are concerned with using the disassembler to produce an ASCII source code file (also called a text file). The first phase is to identify the type of data the program is composed of. Programs consist of machine instructions and various types of data. But which is which? You must follow the logic to tell. The only technical difference is that machine instructions are executed while data is referenced. It is

quite possible for instructions in one part of a program to be data to another part. With the different segments of the Intel 8086 this is not very common. But it is certainly possible.

Once the program has been divided into instruction and data areas, the next phase begins. This is the process of identifying the different logical sections. This is usually the most difficult and time consuming part. It is not easy to understand what purpose a sequence of instructions has, but with persistence this can be done.

The final stage involves generating an assembler source file and getting it to reassemble properly. Disassemblers are only "human." Their output may assemble without error but it probably won't be a byte-for-byte copy of the original file. Some "touch-up" will be required to rectify such things as long and short jumps. While you are at it, you could clean up the comments and "pretty up" the source file.

Note before I go on with examples, I am assuming that the disassembler used will maintain a table of addresses that have been referenced. It that way it would "know" if a label should be included on a particular line or not. Since this table, or label pool, can only be built by looking at the entire program, it will be incomplete when you start and become progressively more accurate as the program is disassembled.

Identifying Data Types

There is a real knack to separating the code into data and instruction areas. Assemblers that mark questionable instructions go a long way in making this easier. What are questionable instructions you ask? Well, these are either 1) invalid instructions (those that the 8086 processor does not understand), 2) meaningless instructions, or 3) very rarely used instructions.

Initially I will assume that the entire code segment is made up of instructions. The data segment (EXE type programs only) would all be pure binary data. However, most of the time this is not the case. It is very common to find character strings imbedded within the code as well as normal data areas. If questionable instructions exist, examine the lines above and below to determine where the instructions end and data begins. Of course it is possible that instructions seem questionable at first and later turn out to be correct. In fact this happens all of the time.

There are five basic rules that can be used to determine data area types. When you go to identify data areas, make sure these rules have been satisfied. If not, be very suspicious.

• Rule 1—The instruction preceeding a data area must be a transfer (jump, call,

Figure Ia, Typical Display Of Partially Disassembled Program

```
127D:BE5011      _          MOV   SI,#L1150H    ;
1280:E822FC                 CALL  L0EA5H        ;
1283:E9DB1B                 JMP   L2E61H        ;
1286:55          L1286H     PUSH  BP            ;
1287:6E          ?          DB    06EH          ;
1288:6B          ?          DB    06BH          ;
1289:6E          ?          DB    06EH          ;
128A:6F          ?          DB    06FH          ;
128B:776E                   JA    L12FBH        ;
128D:207665                 AND   [BP]+65H,DH   ;
1290:7273                   JC    L1305H        ;
1292:69          ?          DB    069H          ;
1293:6F          ?          DB    06FH          ;
1294:6E          ?          DB    06EH          ;
1295:206F66                 AND   [BX]+66H,CH   ;
1298:205475                 AND   [SI]+75H,DL   ;
129B:7262                   JC    L12FFH        ;
129D:6F          ?          DB    06FH          ;
129E:0D0A00                 OR    AX,#L000AH    ;
12A1:FF          ?L12A1H    DB    0FFH          ;
12A2:9C          L12A2H     CBW                 ;Convert byte (AL) to word
12A3:2E803EA112FF           CMP   CS:[L12A1H],0FFH;
12A9:7404                   JZ    L12AFH        ;
12AB:9D                     CWD                 ;Convert word (AX) to dbl w
CS:: Labels= 492/23%, Types=  21/  4%,   0 cmnts  No Edit 10/ 3/87 10:20:35
```

Figure Ib, Typical Display Of Partially Disassembled Program

```
127D:BE5011      _          MOV   SI,#L1150H    ;
1280:E822FC                 CALL  L0EA5H        ;
1283:E9DB1B                 JMP   L2E61H        ;
1286:556E6B6E6F77 L1286H    DB    'Unknown version ';
1296:6F6620547572           DB    'of Turbo',CR,LF,0;
12A1:FF          L12A1      DB    0FFH          ;
12A2:9C          L12A2H     CBW                 ;Convert byte (AL) to word
12A3:2E803EA112FF           CMP   CS:[L12A1H],0FFH;
12A9:7404                   JZ    L12ACH        ;
12AB:9D                     CWD                 ;Convert (AX) to dbl word
12AC:C3          L12ACH     RET                 ;
12AD:2EC606070809 L12ADH    MOV   CS:[L0807H],#09;
12B3:C3                     RET                 ;
12B4:C4C5                   LES   AX,BP         ;
12B6:C6C7C8                 MOV   BH,C8         ;
12B9:E8C6FC      L12B9H     CALL  L0F82H        ;
12BE:8B4616                 MOV   AX,[BP]+16H   ;
12C1:A38A01                 MOV   [L018AH],AX   ;
12C4:8B4604                 MOV   AX,[BP]+4     ;
12C7:A38C01                 MOV   [L018CH],AX   ;
12CA:1E                     PUSH  DS            ;
12CB:C516AA11               LDS   DX,[L11AAH]   ;Load DS:DX with 32b pointe
12CF:B010                   MOV   AL,#10H       ;
12D1:B425CD21               MSDOS _SIVEC        ;Set vector.
CS:: Labels= 492/23%, Types=  21/  4%,   0 cmnts  No Edit 10/ 3/87 10:20:35
```

Figure Ic, Typical Display Of Partially Disassembled Program

```
127D:BE5011      _          MOV   SI,#L1150H    ;
1280:E822FC                 CALL  L0EA5H        ;
1283:E9DB1B                 JMP   L2E61H        ;
1286:556E6B6E6F77 L1286H    DB    'Unknown version ';
1296:6F6620547572           DB    'of Turbo',CR,LF,0;
12A1:FF          L12A1      DB    0FFH          ;
12A2:9C          L12A2H     CBW                 ;Convert byte (AL) to word
12A3:2E803EA112FF           CMP   CS:[L12A1H],0FFH;
12A9:7404                   JZ    L12ACH        ;
12AB:9D                     CWD                 ;Convert (AX) to dbl word
12AC:C3          L12ACH     RET                 ;
```

(continued)

interupt, or return). Conditional jumps would not be allowed.

- Rule 2—The first instruction in an instruction area must have a label unless the preceeding data area was an argument to a call or interupt instruction.
- Rule 3—An absolute transfer of control (jump or return) may be followed only by a labeled instruction or a labeled data area.
- Rule 4—For the type of data to change (from instructions to data or from ASCII data to 16-bit address data etc.), the first line of the newer type must have a label.
- Rule 5—ASCII character data (including carriage returns, line feeds, etc.) must either begin with a character count byte (or word) or it must end with with a special byte. It is common within MS-DOS applications that character strings end with a dollar sign. This is the way the console output and printer output functions know the end of a string. Assembly programmers also like to use null characters (value of zero) as an end of string mark. The Intel 8086 processor can easily detect these.

For purposes of an example, Figures Ia through Ic will be used. This is fairly typical of the kind of code you will encounter. But be forewarned, by its very nature assembly code can be very obscure. If the programmer wishes, it could be extremely difficult to decipher.

Refering to Figure Ia, note how several lines have been marked as "questionable" (note, MD86 inserts a question mark in front of the label field). Here it is obvious that the lines following the jump instruction at address 1283 cannot be instructions. The PUSH instruction at address 1286 is erroneous because of rule #1. Notice how most of the bytes following address 1283 have a value in the range 20 to 7E (hex). It is quite possible that this area consists mainly of ASCII characters. But where does this area end? Rule #2 says we should look for the next valid instruction line containing a label. In this example we find this at address 12A2. A word of caution here. Since we may not have disassembled the entire program, the label pool may be incomplete. It is then possible that at this time an instruction does not have a label. We need to be cautious in the application of rule #2.

As a first step, we change the area from 1286 to 12A1 from instructions to ASCII characters. The code now makes more sense (see Figure Ib). But now notice address 12B4. This instruction does not have a label and yet it follows an unconditional transfer (return) instruction. Rule #3 says this is not correct. Now it could be that there should be a label here and we have just not disassembled the section of code that references it, but the instructions don't look right do they?

```
12AD:2EC606070809 L12ADH   DB    2EH,0C6H,6,7,8,9,0C3H;
12B4:C4C5C6C7              DB    0C4H,0C5H,0C6H,0C7H;
12B8:C8                    DB    0C8H          ;
12B9:E8C6FC        L12B9H  CALL  L0F82H        ;
12BC:8B4616                MOV   AX,[BP]+16H   ;
12BF:A38A01                MOV   [L018AH],AX   ;
12C2:8B4604                MOV   AX,[BP]+4     ;
12C5:A38C01                MOV   [L018CH],AX   ;
12C8:1E                    PUSH  DS            ;
12C9:C516AA11              LDS   DX,[L11AAH]   ;Load DS:DX with 32b pointe
12CD:B010                  MOV   AL,#10H       ;
12CF:B425CD21              MSDOS _SIVEC        ;Set vector.
12D3:1F                    POP   DS            ;
CS:: Labels= 492/23%, Types=  21/  4%,   0 cmnts  No Edit 10/ 3/87 10:20:35
```

Figure II, Sample Disassembly of EXE2BIN.EXE

```
0000:1E                _       PUSH  DS            ;
0001:33C0                      XOR   AX,AX         ;
0003:50                        PUSH  AX            ;
0004:B430CD21                  MSDOS _GETVER       ;Get DOS version number
0008:3C02                      CMP   AL,2          ;
000A:7D13                      JGE   L001FH        ;
000C:BB3900                    MOV   BX,#L0039H    ;
000F:8EDB                      MOV   DS,BX         ;
0011:BA5B01                    MOV   DX,#L015BH    ;
0014:0E                        PUSH  CS            ;
0015:1F                        POP   DS            ;
0016:B409CD21                  MSDOS _OUTSTR       ;Display string at (DS:
001A:06                        PUSH  ES            ;
001B:33C0                      XOR   AX,AX         ;
001D:50                        PUSH  AX            ;
001E:CB                        RET                 ;*** FAR RETURN ***
001F:BE8100            L001FH  MOV   SI,#L0081H    ;
0022:BB3900                    MOV   BX,#L0039H    ;
0025:8EC3                      MOV   ES,BX         ;
0027:8B1E0200                  MOV   BX,[L0002H]   ;
002B:E82B01                    CALL  L0159H        ;
002E:7308                      JNC   L0038H        ;
0030:06                        PUSH  ES            ;
0031:1F                        POP   DS            ;
0032:BA9001                    MOV   DX,#L0190H    ;
CS:: Labels= 164/ 7%, Types=   3/  0%,   0 cmnts  No Edit 11/ 5/87  3:12: 2
```

Figure III, The Program Segment Prefix Summary

Offset	Contents
0002	System memory size in paragraphs (16 byte blocks). This is a 16 bit integer.
000E	Control-C exit address. First 2 bytes are offset and second 2 bytes are the segment.
0012	Hard error exit. 2 byte offset and 2 byte segment.
005C	Unopened file control block for first file specified after command. Only valid if a path is not specified.
006C	Unopened file control block for second file specified after command. Only valid if a path is not specified.
0080	Entire text string that follows the command. The first byte is a character count. Note redirection information is not passed on to the program (it is stripped first).

The hex sequences 06, 07, 08, 09, and C3, C4, C5, C6, C7, C8 would not likely be instructions (although obviously possible). It looks more like numbers or data. In fact the whole area from address 12AD up to 12B8 does not look like instructions at all. Most probably this is just a data area containing numerical values. And 8-bit values at that. If they were 16-bit values (or addresses), they would be way beyond the bounds of our code.

This area is then changed into binary data (8-bit) from instructions. Figure Ic shows what the screen looks like now. Compare this with Figure Ia and you can see the improvement. In this way areas of the program are disassembled one section at a time. Progress at first seems slow I realize, but after a while the pieces start to fit together. As you begin to understand these small portions, the remainder of the program becomes that much easier. You are well on your way to a useful source file.

For EXE type programs, there is a separate data segment to worry about. While this probably does not contain instructions, it is still necessary to determine if there are any address references stored here. If there are, then they should be identified as such so they can be entered into the label pool.

In some cases tables of addresses can be spotted easily. If most of the addresses are close (within a few pages) then you will see similar hexadecimal values every other byte. For example:

```
1234:017F097F0F7F L1234H   DB    1,7FH,9,7FH,0FH,7FH;
123A:137F4F7F1080          DB    13H,7FH,4FH,7FH,10H,80H;
```

When these areas are changed into 16-bit addresses then they appear as follows.

```
1234:017F097F0F7F L1234H   DW    7F01H,7F09H,7F0FH;
123A:137F4F7F1080          DW    7F13H,7F4FH,8010H;
```

The contents of these areas are then added to the address label pool. When disassembled, these areas will have a label to let you know that they are referenced somewhere.

Notice how the first address of this table has a reference. Rule 4 indicates that this is required. However this is not strictly true. It is possible that the beginning of this area is implied by the end of the previous structure. One common approach is to have a sequence of flag bytes that is followed by a corresponding address table. Because the program "knows" how long the leading byte table is, it then knows the start of the address table.

More than likely, address references present within the data segment refer to offsets within the code segment. If your disassembler does not make this assumption, then the label table may have to be fixed by hand. As much as possible, you want to avoid contamination of the label table with erroneous references. This just makes life more difficult.

Understanding the Code

This is the part you have been waiting for. The real guts of the job! You have now separated all data from instructions, but what do the instructions mean?

The Intel 8086 executes instructions in a logical order; the order chosen by the programmer. To truly understand the function of the instructions you must know how they are executed. For example, just knowing the instruction

```
123A:2C07              SUB   AL,7
```

will subtract 7 from the contents of register AL is not very helpful. However, if the surrounding instructions were

```
1234:8A07              MOV   AL,[BX]      ;
1236:3C3A              CMP   AL,':'       ;
1238:7202              JC    L123BH       ;
123A:2C07              SUB   AL,7         ;
123B:2C30    L123BH    SUB   AL,'0'       ;
123D:8807              MOV [BX],AL        ;
```

you then have the feeling that register BX is pointing to one or more bytes. And if these bytes are greater than the digit 9 (the character ":" is just past the digit "9" in the ASCII character set) then 7 is subtracted. Looking 7 past the "9" digit in the table of ASCII characters you find the letter "A". Then in either case the value of the digit "0" is subtracted. In other words, if register BX were pointing to an "8", then this would be replaced with the binary value of 8. If, however, BX points to the letter "C", it will be replaced with the value 12. So this is just converting a hexadecimal digit or digits from ASCII to binary. Well of course! We "know" this program asks for hexadecimal values and has to interpret them because in this case we are looking at a DEBUG.

Because the processor executes instructions in a certain order, we must examine them in that order. This might seem obvious (and in the above example it is) but in many cases it is not easy to determine the way in which instructions are executed. Consider the following code.

```
1234:E83033            CALL  L4567H       ;
1237:0130              ADD   [BX+SI],SI   ;
1239:337200            XOR   SI,[BP+SI]+0 ;
```

The ADD instruction following the CALL is not actually executed at all. By looking at the routine at address 4567 we find that the byte following the initial CALL is just a parameter. This byte gets used and the return will be to the following address (1238). We would not have been able to tell this if we hadn't looked at the instructions in the same order the processor does. The code should actually appear as follows.

```
1234:E83033            CALL  L4567H       ;
1237:01                DB    1            ;
1238:3033              XOR   [BX+DI],DH   ;
1239:7200              JB    L123BH       ;
```

When you pick apart even a small section of code you should enter a few comments and add a label name if you can. Then you won't have to reinvent the wheel the next time you look at this code (and you will look at it more than once!).

This process is going to be very laborious. It takes many instructions in assembly language to accomplish seemingly trivial functions. Like the simple BASIC statement "LET A(1,2) = B + C^2" may take thousands of instructions and involve many subroutines. But all is not lost. Because you know how the program executes (at least in a gross sense), you will be able to tackle small portions of it at a time.

Any information you can get your hands on will help. User manuals, especially reference manuals, are a valuable source of information. Some go so far as to include memory maps and descriptions of internal data types. Take TURBO Pascal for example, the manual is a real gold mine!

A bottoms up approach has proven to be the most useful when disassembling a program. Start from the lowest level. Look for the operating system interface. The reason is that these are well defined and have a specific calling sequence. Some disassemblers recognize the MS-DOS system calls and use more meaningful representations. For example the instructions

```
1234:B409              MOV   AL,9         ;
1236:CD21              INT   21H          ;
```

is replaced with a single macro instruction

```
1234:B409CD21    MSDOS _OUTSTR   ;Display string at (DX)
```

In this way you can identify the lowest level routines. Those that write characters to the screen or read the keyboard. How about opening and closing files and input and output from the communications ports? Generally these are short subroutines (<100 lines) that you can comprehend. Try to find as many of these routines as possible and give each one a name that will help you to remember what it does. Also toss in as many comments as you can.

Once the lowest level routines have been worked on, the next

higher level becomes easier. Now you can find those routines that read and write to file buffers without worrying about all those instructions required to actually get the data out to the disk.

In this way the program gradually starts to unravel and before you know it you will actually understand how the programmer was able to write it.

Execute files (those with the extension EXE) introduce a whole set of additional problems. Not the least of which is determining actual physical address for instructions. You see, the Intel 8086 constructs the physical address at run time from a segment register and an offset. The relationship is:

```
physical address = segment*16 + offset
```

Because each register is 16 bits long, there is the possibility of tremendous overlap. An offset of 100 into segment 1234 is the same as offset 110 into segment 1233. To further complicate matters, the segment registers can be changed at will. Thus when an instruction is executed, the contents of the segment registers (which may have been defined who knows where) are of vital importance. The more segment registers are modified within a program, the tougher the job of disassembly is.

As an example of a typical execute program, let's look at EXE2BIN.EXE. Within the first few instructions we see the code shown in Figure II.

Let's look at this code for a second. We see that almost the first action of this is to call MS-DOS and find out what its version number is. If this number is greater than or equal to 2 then this jumps to offset 001F. So the code between 000C and 001E is only executed if the version number is less than 2. Following the jump instruction, the next two instructions initialize the data segment register (DS) to 39 hex. That means that further references into the data segment will get to physical address 390 hex + offset. The next instruction loads the DX register with the value 15B hex. Now if we take a quick look at address 4EB hex (390 + 15B = 4EB) in our code we will find the start of the ASCII message "Incorrect DOS version$". A quick note, normally these addresses (ie 4EB) will be relative to the start of the data segment within the EXE file, and the code segment follows this immediately. Thus we have to look at 4EB – data_segment_size within our code. But for EXE2BIN.EXE, the data segment size was zero so we can look directly at address 4EB. Now the two following instructions are very curious. By executing the PUSH CS and POP DS we will effectively reset the data segment register to the code segment register, or zero within our file. Thus the call to MS-DOS function to display an ASCII character string will try to get the characters from offset 15B instead of 4EB. This is a definite bug in EXE2BIN.EXE! The PUSH and POP instructions should not be there. Even the best programs can contain bugs. Don't be too alarmed when you run into one.

Moving on, at addresses 22 and 25 we see that the extra segment register (ES) is being set to 39 hex just like the data segment register was set. This should give us a real strong indication that at address 390 hex (or a few bytes beyond) we will find the start of a data area within our code. This will help us later on.

One further note, when MS-DOS executes an EXE type program, it initializes the data segment and extra segment registers to point to an area called the Program Segment Prefix (PSP). This area contains many useful items that the program will need. So prior to changing these registers, the program will examine this area for those items it needs. Figure III lists those items that are of most interest to us. Refer to reference 1 for a more complete discussion of this area.

Polishing the Source Code

Sooner or later you will come to the point where you must abandon the disassembler. It has done its job but now an editor would be better suited to working on the files.

Once you generate a source file you can try assembling it. There will undoubtedly be many areas where the assembler will complain. Segments may be defined in the wrong order or some external references are not defined at all.

Get yourself a good screen oriented editor. One with virtual memory support is vital. Assembly programs tend to be very large and it will be a real pain if you have to break it into small pieces because your editor limits the code to 64k. You are going to especially need global search and replace functions. WordStar, although rather slow, does work fine for this type of work as long as you don't use document mode.

The disassembler will probably insert too many data type pointer override instructions. These are the WORD PTR and BYTE PTR sequences you see all over the place. Assemblers, like MASM, do not require these overrides if the types already match. That is, if a value is referenced as a 16-bit word and it has previously been defined as this type, then an override is not required. If the disassembler is not sure that these conditions have been met, WORD PTR (or BYTE PTR) will be inserted. One of the first things I like to do is to remove these phrases where they are not needed. For me, they just clutter the code.

EXE type files pose the biggest challenge to the disassembler. The assembler will certainly complain about some aspect of the way the different segments are handled.

One of the difficulties disassemblers have is what to do about a numeric reference. MD86 rather simplemindedly inserts a variable reference for each occurrence unless the address is outside of the code boundry? And is the reference really to an address or is it just a constant? Answers to these questions are difficult and each disassembler handles them differently. These issues must be dealt with before you can assemble the program. Note that MASM is very sticky about variables and constants. If a label is defined as a constant (using EQU), then MASM issues the message `error 56: No immediate mode` when this label is referenced as a variable. This is telling you that the EQU instruction must be changed to a DB or DW.

Deciphering More Obscure Code

In the good old days when memory was expensive and processors had a limited address range, assembly programmers delighted in seeing how much they could squeeze into small spaces. This tendency has lessened somewhat with the newer processors and cheap memory, but you will still find some real funny looking code.

Consider the following which was found at the start of a disk input and output routine.

```
1234:F9          STC
1235:73F8        JNC   L122FH
1237:B80100      MOV   AX,1
123A:7304        JNC   L4567H
123B:7304        JC    L7654H
```

Wait a minute, you say. How can you have a set carry instruction (STC) immediately followed by a jump on no carry (JNC)?. There must be something wrong. No one writes code like that! Actually this code is correct. Since the jump on no carry is never executed, the destination byte is always skipped if the instructions are executed in the order shown. However, the programmer sometimes jumps directly to address 1236 which is in the middle of the jump instruction. In this case, the displacement is executed and this becomes a clear carry instruction (the F8 byte). What happens is that the routine has two functions that are very similar (like keyboard input with and without echo) and the state of the carry flag is used to determine which function is desired. A jump to address 1234 does one thing and a jump to 1236 does the other. Very sneaky!

Or how about this piece of code.

```
1234:40             INC   AX
1235:40             INC   AX
1236:40             INC   AX
1237:40             INC   AX
1238:40             INC   AX
1239:E82B33         CALL  L4567H
```

It doesn't make sense to have that many increment instructions in a row. Or does it? Actually this is part of an error handling routine. The idea is to load the AX register with an error number and call the routine at 4567 to print out a message based on the error number. To display error number 1, then the programmer writes the code

```
2345:31C0           XOR   AX,AX
2347:E8EEEE         CALL  L1238H
```

To display error message number 4, then the call goes to address 1235 instead. For this particular procedure, the AX register always contains a zero (it is used as an error flag) and so the XOR AX,AX instruction can be eliminated. Then this requires only a three byte call instruction to flag an error condition (instead of the usual five bytes). Some programmers go to great lengths to save a few bytes of code!

When you discover this, you immdediately enter some appropriate label names. The code now looks better.

```
1234:40       Error5    INC   AX
1235:40       Error4    INC   AX
1236:40       Error3    INC   AX
1237:40       Error2    INC   AX
1238:40       Error1    INC   AX
1239:E82B33   NoError    CALL  L4567H
```

Making Use of the Generated Source Code

Now that you have generated the source code and it has been documented more of less to your liking, what do you do with it? Although you probably should have asked yourself this before you started this project, here is a list of the major uses for the source code to a program.

- Using the source code you can fix those "bugs" that prevented you from fully utilizing the product.
- Set those default parameters to what you want.
- Modify the program to include features that you (and perhaps others) want to see.
- Learn how the programmer did those wonderful things. Make your programs work just as well using these techniques.
- Make some money from your labor.

Because you have converted the program from machine code to an ASCII file suitable for an assembler, you can now edit the file making any changes you wish. Then you reassemble and link (or load) the program. This new copy now functions the way you want it to.

If one of your reasons for producing the source code was to learn how the program functioned, then the printed listing is probably the most important result. While the process of disassembling already gave you great insight into the program as a whole, now you will be able to study that particular aspect that intrigued you.

Another interesting, but often overlooked, use for the source code is that other people may have similar interests to yours. Maybe you could sell it? This is a sticky area because the source code is composed of instructions whose order (ie, the executable part of the program) is owned by the author, and your added comments and label names. There have been a few examples of disassembled programs being published (for example the Timex/Sinclair ROM) but these most likely required the consent of the original author.

Regarding your work, you would certainly be allowed to sell your label names and comments but probably not the instructions. After all if you sold the instructions then you are, in effect, selling the program and would owe the author royalties. But all is not lost as there is a way out of this mess.

If you consider that every line of the assembly source file looks like this:

```
<-- #1 ---> <---------- #2 -------> <---------- #3 -------> <--- #4--->
{LabelName} {processor instruction} {instruction arguments} {Comments.}
```

where each field could be absent, then you could extract the portions that you have created (#1 and #4) from the line and eliminate the portions that the program author created (#2 and #3). The extracted portions could be put into data files and sold. Now in order for a customer to use this data (your label names and comments), he will have to come up with the instruction and argument fields for each line. This means he must already have the program. Since even a rudimentary disassembler has no trouble producing the processor instructions for a program, you will have to come up with a disassembler that knows how to combine your label names and comments with the disassembled instructions. This is not a trivial task, but it's not that difficult either. What you come up with is a program that can reconstruct the source code on the customer's computer system.

In this way you would only be distributing your label names and comments along with a simplistic disassembler that knows how to put the pieces back together again. And, importantly, the author is not cheated out of any royalties that are due because the user must already have a copy of the program. Which hopefully was obtained properly.

The results of this fancy foot work is that the customer has to wait a short while for the source code generator to produce the final files. The generator for TURBO Pascal takes about twenty minutes to produce 21,000 lines of assembly code. Not a big inconvenience considering the immensity of this task.

These knowledgable, highly specialized disassemblers are what I call Source Code Generators. For various products the work has already been done for you. C.C. Software has these available for CP/M 2.2, CP/M 3, and TURBO Pascal v 3 (Z-80 and 8086). And more are in the works.

References:

1) *MS-DOS Developer's Guide*", John Angermeyer and Kevin Jaeger, Howard W. Sams & Co, 1986

2) *Peter Norton's Assmebly Language Book for the IBM PC*, Peter Norton and John Socha, Prentice Hall Press, 1986

■

Real Computing

The National Semiconductor NS32032

by Richard Rodman

Start off with a tricycle. It's nice, basic transportation, not too fast, but predictable. But before you know it, users are complaining that it's too slow. So, you graft onto its rear a motorcycle engine, transmission and rear wheel. Faster, but you underestimated the users' demands, so onto the right side, you graft half an automobile with two wheels. Grotesque, right? This image illustrates how both Intel and Motorola have grown their 32-bit processors through evolution.

It's important to remember that computer history didn't begin with microprocessors. The 8008 and the 6800 were drastically downsized computers-on-a-chip, embodying a few design concepts of minicomputers of the day. Microprocessors then evolved from these starting points into more complex designs along their own growth paths. The 386 processor of today still has a strong resemblance to the 8008.

Someone could have asked, instead of trying to redevelop the computer along an independent line of development, why not try to duplicate an existing, successful computer architecture, proven in the field? Someone, in fact, did ask. National Semiconductor developed its 32032 family of chips, not as a SC/MP with 32-bit registers, but as an emulation of DEC's highly successful and powerful VAX architecture.

With what result? With the result that, like the VAX, the National Series 32000 processors have one instruction set for the family—all the chips are completely object-code compatible. There are no "gotchas" such as required word-alignment, separate "address" and "data" registers, or addressing modes that only work on certain registers or instructions. It's a clean instruction set, a joy to use from a software standpoint. How often do you have to look at an instruction reference manual? Intel programmers take it to bed. Motorola programmers take it to work. National programmers leave home without it.

Some examples of NS32 instructions are shown in Figure 1.

So, the National series maximizes the value of the programmer's time. "But how can I get to use this great instruction set?" you might ask. Alas! There are no ready-made glitter boxes available from Toys-R-Us or K-mart sporting an NS32 processor. Thus, the NS32 experimenter is forced to venture into the dark mysteries of hardware construction.

```
                    Real Computing

ADDD    R0,R1     Add double (32 bits) register R0 to R1.

CMPB    -6(FP),R1 Compare the byte 6 bytes below the frame
BLT     LAB1      pointer, and branch to LAB1 if it is
                  less than R0.

ADDW    LAB1,LAB2 Add the value at LAB1 to the value at
                  LAB2 (note - no registers used!)

MULW    TAB1(R0),-12(FP)
                  A similar example - all addressing modes
                  can be used in most operands of most
                  instructions.

SAVE    [R0,R1,R5-R7]
                  Compare to MOVEM.L [A0,A1,D2-D5],(A7)-.
                  Which is clearer?
```

Figure 1: Examples of NS32 instructions.

Hardware Options for the Experimenter

The least painful way to get into NS32 computing is to obtain a PC coprocessor board. This is a board with a CPU and a meg or more of RAM which plugs into a PC. The PC's Intel processor controls the disk drives, screen and keyboard, becoming, in effect, an I/O processor for the real CPU. Several of these are available, most notably from Definicon Systems and Opus Systems. They usually run Unix and sell for $1500 and up.

Another PC coprocessor effort, which was intended to sell for less than $1000, was the PD32. This board was designed by Dave Rand and George Scolaro and described in *Micro Cornucopia* magazine. I don't believe it is available anymore, but you might find one used.

Next down the line are Multibus boards, produced by National Semiconductor themselves and others. These boards tend to be costly and not immediately useful, but with other Multibus boards, you can build up a system.

CompuPro produced an S-100 CPU board, the CPU-32016. It was not too successful (they had no software to run on it), but it's a very well-designed board, using the full 16-bit mode of the bus. Last I heard, they still had some available which they were selling off at $95 apiece, a bargain.

Of course, there are complete systems like the Sequent and exotic VME bus boards you can buy, but these are really big-ticket items that cost mucho denaro.

For the more serious or budget-conscious, or both, there are the Designer's Kits, available from your local Hamilton-Avnet. These include a CPU, either a 32016 or a 32032, and a floating-point unit (FPU), memory management unit (MMU), interrupt control unit (ICU), timing control unit (TCU), a ROM monitor/debugger/editor/assembler, a decoding PAL, some books and a schematic. The 32016 version is $60, and the 32032 version is $75. You'll need to wire-wrap a circuit, and you'll need some sockets, static RAMs, and glue chips.

CPU's in the Series

Before I go on, let's get the various family members straight. First, there's the 32032, the basic 32-bit CPU of the series. It's available at 6, 8 and 10 MHz. Then there is the 32016, which has a 16-bit bus, and the 32008, which has an 8-bit bus.

Moving up, there is the 32332, which has an address bus expanded to 32 bits (the earlier versions only have a 24-bit address) and can run up to 15 MHz, hitting roughly 2 MIPS. And there is the brand-new 32532, which has an on-chip MMU and is capable of

about 10 MIPS. I cite MIPS figures hesitantly. Don't compare MIPS figures with smoke-and-mirrors MIPS figures given for RISC CPU's. To convert RISC MIPS to VAX MIPS, you have to divide by 10 to 15 or more. To be fair, RISC CPUs have done us a great service by greatly advancing the state of the art in contriving benchmarks.

The NS32 CPU's are all object-compatible: PROMs burned for one CPU will work on all the others.

Software Options for the Experimenter

If you get a PC coprocessor with Unix, you're all set for whatever you have in mind. If you get anything else, though, things don't look so peachy. The Designer's Kits come with a Tiny Development System, which is a ROM monitor with debugger and a tiny editor and assembler. It's not suitable for any very complex work.

Between the two extremes is a yawning chasm that a loosely associated group of intrepid young NS32 experimenters is seeking to bridge. (Well, some are young, anyway.) Neil Koozer wrote the Z32 assembler, which runs on a Z80 under CP/M or under MS-DOS using Z80MU: He has been working on 32HL (see *TCJ* #30), an integrated development environment for the NS32.

There are a couple of other assemblers available, plus Small-C and a full C compiler. Work has been proceeding on a simple CP/M-, MS-DOS-like operating system for the NS32. This OS will be mostly written in C and will be distributed in source code form.

Once we have a minimum system running, it will be freely distributed. I'd like to make it a lot like VMS, but Marc Lewis wants to make it like AOS, and Brian Haug wants it to be like Unix. It'll be great! We'll make it accomodate user-written command processors, so that everyone can tailor it to his or her own desires, but still be able to interchange programs and data. Please, though, send me your changes so that they can be shared with everyone.

The PC 360K diskette format has emerged as the world's predominant data interchange medium. More computers can read this format than all others combined. Therefore, nearly all experimenter's operating systems use this format, such as Hawthorne Technology's K-OS-1 for the 68000. The format is actually very good for technical reasons, too, but I won't go into that here.

Why This Column?

So, the purpose of this column is threefold: first, to expose more people to the power of the NS32 architecture; second, to support the growing number of NS32 experimenters out there, both in hardware and in software; and third, to get more people involved in the public domain operating system project.

Yes, we're giant-killers. But what we keep finding is that while the giants certainly are slow, they're not all that big after all.

Where to write—where to call

CompuPro/Viasyn Corp.
26538 Danti Court
Hayward CA 94545-3999

Definicon Systems Inc.
1100 Business Center Circle
Newbury Park CA 91320

Opus Systems Inc.
20863 Stevens Creek, Building 400
Cupertino CA 95014

Richard Rodman
1923 Anderson Road
Falls Church VA 22043
BBS (Virginia—East Coast)
703-847-2951

■

S-100 EPROM Burner

(Continued from page 18)

this. Talked to someone at Digital Research (very helpful), and he thinks that request is just impossible. Ignored it and everything seems fine.

That's it. Took a couple days, but one day would be plenty for anyone who actually has some idea of what he is doing. The only trouble I can see with the board has been a certain lack of consistency in the connection it makes in the S-100 backplane. At least, that appears to be the cause of an inability to get correct voltage measurements until I pushed the board around a bit. Later, when watching the LED during EPROM programming, the same thing was apparent, but then if you are seeing the LED light, everything is fine. If I don't, I just "push the board around a little." The board certainly appears to be just fine, and I'm sure if there is anything wrong it's from the rough handling I gave it during construction and during testing. You perform a resistance test before starting to see if the board is generally all right, and that was fine.

The board comes with a remote board and an EPROM burning program written in C that seems quite luxurious. I've already "captured" the contents of my printer ROM for later manipulation. That remote board is quite the thing. You plug in your S-100 board to which you've attached a 40-wire ribbon cable, hang the cable through a convenient hole somewhere, then install the remote card (with its ZIF socket(s)) anywhere that is convenient for you.

Through use of configuration headers (little 16-pin machined pin sockets with templates on them), this system programs any EPROM from a 2716 through the 27128, as long as it can be programmed with either 21 or 25 volts. Noticed just in time that the 2764s and 27128s that JDR sells are 12-volt chips, so watch out. Found 21-volters in other ads, so they're not rare.

By the way, I recommend ZIF sockets (I used TexTool) for both the programming socket (a 28-pinner) and the configuration socket. I've bent all the configuration leads and even broken one taking the headers out of the configuration socket. I've seen something advertised for pulling these things, so maybe that would be wise, too. The ZIF programming socket is a dream. Just get the EPROM somewhere near the socket and it comes out and gets it from you. Simplicity itself.

Digital Research is still in business, incredibly enough. Get a "catalog" from them at P.O. Box 381450, Duncanville TX 75138 (214) 225-2309. ■

Computer Corner

(Continued from page 40)

anybody, as it gives examples of how to talk to hardware. For me it will be interesting to talk to him next year and see how his self publishing is doing. I have been thinking of doing a similar project myself, but more along the lines of Forth code and Forth ENGINES.

Forth ENGINES, or CPUs that run Forth directly, are actually very powerful devices which many organizations have not been considering for projects. I feel a good tutorial type text, showing interfaces, projects, and Forth code would be of interest for many of our readers and the industry. My biggest problem with the project is keeping up with all the new engines and vendors. Unless the market crashes soon, there will be a lot of new Forth based products hidden away in new markets. With that I guess I should take my cue and start writing.

NEXT

No, this is not a Forth concept, just my way of ending this corner and hoping next time to be back on my regular topics, whatever that happens to be. To contact Bruce and get his book, write "eisys", 1009 North 36th Street, Seattle, WA 98103. ■

S-100 EPROM Burner

Building the Digital Research Board

by Michael Broschat

I'm one of those people who when he felt he had to have a personal computer back in 1983 listened to a "friend of a friend" and ended up with an S-100: a Sierra Data Sciences SBC-100 with " drives and a Qume 102 terminal, to be exact. Now, I realize the dealer must have been ecstatic to be getting rid of what was increasingly appearing to be a white elephant.

I suppose that since my only computing experience had been some very mild word processing on a university mainframe, I could have done worse. Now don't you hardware types get riled up. I know very well that at that time the concept of the S-100 was "right" and that in some sense it still is. But not for people like me, whose only need for the personal computer was to write very non-technical things. Boy, have times changed. Five years later I am burning EPROMs (after disassembling my BIOS) and doing technical translations from the Chinese.

But I've had so much to learn, I am way behind everyone else. Just when I am getting comfortable with my equipment, think I know something of how it works, and am more than mildly addicted to ZCPR, everyone else is bailing out. There can't be more than about 12 of us left doing 8-bit things, and most of them seem to be back east.

Anyway, I've known for quite some time that there are changes I want to make in my system, but just haven't gained enough skill to manage them. I bought a Magnum Digital disk controller board a couple years ago to gain 5" drive capability, but have never been able to get it to work on my existing system. A couple years of various experiments have shown that the new board is conflicting somehow with the SBC (not because of shared ports), and the only way out seems to be to keep the SBC from programming the various Zilog chips (zillions of interrupts) before booting the system. Could be something else, of course, but haven't got any further yet.

So, how to do that? I finished a disassembly of my system BIOS (Sierra Data only supplied code for a "user BIOS"—console and printer I/O, etc.) some months ago, and am reasonably sure now how the boot PROM and system BIOS are interacting to create my problems. I commissioned an attempt to reprogram the contents of the boot PROM, but that failed, and I think I know why. Programming the EPROM was not free (cost me about $20), and I figured that before I get successful things could get expensive, so looked again at the Digital Research of Texas ads in various hacker rags and took a chance on their EPROM burner at about $40 for the bare board.

I did an stereo amplifier kit once (Dynaco), but that's been the limit of my experience. I knew that with a bare board I would be responsible not only for understanding the instructions, but also for finding the parts in the first place. Did the best I could from the parts list and a JDR Microdevices catalog. Made some mistakes and succumbed to a couple failings in the documentation, but all in all it has turned out pretty well. I'll say some more specific things later.

I think the reason I'm writing this is because there must be others like me out there, and if you don't act fast to take advantage of deals such as this, they just won't be around much longer. If you are happy with your S-100 and want the ability to reprogram EPROMs (for your computer, your printer, your terminal, who knows what else), look into this. There are other things, too. From the same company I bought 2 megabytes worth of RAM disk boards. What a dream machine I'll have when I can afford the DRAMs! And they also have various RAM/EPROM boards, etc. If I can build a "bare board" project, anyone can.

Some Construction Notes

The resistors don't come labeled. You who know what you're doing knew this already, but I didn't. Someone tipped me off to a $0.60 chart from Radio Shack that helps you figure out what value each one is.

"Machined-pin" was unknown to the JDR representative with whom I spoke, but followed a hunch and ordered the AUGATxx parts (we're talking sockets here). That was correct.

The heatsinks I bought definitely weren't intended for S-100 boards, but a hacksaw straightened them right out. I also used the hacksaw to modify the ribbon header I had incorrectly ordered. Works like a charm.

The parts list calls for a 6-position DIP switch. You'll find that the board was designed for an 8-position switch. The bottom two switches have no effect anyway, so get an 8-position one.

No one could help with the "3-pin 0.1 ctr jumper strip." Didn't know what to do when I got this far. Then I noticed all the resistor and diode leads I had trimmed off up to that point. Perfect. Soldered three of them into the correct places, and there was my jumper strip.

The trim pots (potentiometers) I ordered didn't fit the holes and couldn't be made to. Visited a local electronics store and found the right ones. There doesn't seem to be a way to specify how the leads are arranged. You need ones that are "in-line" rather than staggered.

A 74LS74 is missing from the parts list.

The instructions warn about observing the polarity of the LED. Never could tell this. The one on the main board works fine, the one on the remote is quite dim. Maybe that's the reason?

I still don't know where pin 1 is on all this stuff. A clerk at the electronics store mentioned something about the standard IC pin placement scheme, but I just kept everything "pointing" in the same direction and it works.

Adjustments

I bought the cheapest "multitester" Radio Shack offers, about $10. You must adjust two pots for voltage, then another one for an "active low pulse of 2 ms duration." Found a friend with an oscilloscope, who pushed the right buttons and helped me to make that final adjustment.

One adjustment that seems at fault is to check for 5 volts at pin 28 during a certain step of the set-up program. You never get

(Continued on page 17)

Advanced CP/M
ZSDOS and File Systems

by Bridger Mitchell

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

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

Dawn of a New DOS

Think of it as CP/M 4.0—an all-new, feature-packed, high-performance BDOS replacement for all Z80 computers running CP/M 2.2, ZRDOS, or other compatible DOSs.

ZSDOS is its final name—the cooperative product of Hal Bower, Cam Cotrill, and Carson Wilson that fuses their initially separate efforts. The result is explosive: improved disk function performance, file datestamping with no reduction in program memory, files automatically accessible from other directories, and elimination of some notorious CP/M bugs. Benefiting from Ten Brugge's P2DOS and Carson's first forays into Z80DOS, the finely-tuned final product is fully compatible with BackGrounder ii, NZ-COM and ZCPR34. And the authors' thorough, extensive testing means highest quality; we are unlikely to ever see a ZSDOS 2.2, or 1.9!

ZSDOS is, foremost, an up-to-date DOS. It fully supports the established DateStamper standard, and comes with preassembled relocatable clock routines from the Plu*Perfect Systems library to read virtually all of the popular (and many obscure) clocks. And it breaks new ground, adding BDOS functions to get and set file datestamps as well as to get and set the system realtime clock. In addition to Plu*Perfect's PUTDS, SDD, and DATSWEEP utilities, it is shipped with some nifty new tools that display files sorted by date and that automatically copy datestamps. Best of all, perhaps, is that the "trim" version of ZSDOS includes datestamping within the BDOS with no loss of TPA memory (except possibly for BIOS space to hold a clock routine)!

The "full-up" ZSDOS version adds internal path-searching to the BDOS, enabling *programs* to do what until now only the ZCPR command processor has been able to achieve when loading a command—scan a path of directories to locate a needed file. To do this, it places the datestamping code in a separate small, relocatable module somewhere—in or above the user's BIOS, in NZCOM's user buffer area, or in a resident system extension.

Both versions of ZSDOS provide English-language error messages complete with the name of the associated file, if one. Error reporting is configurable, so that a program can field any error itself, if it chooses. Other significant features include noticeably faster warmboots and disk resets on hard-disk machines.

The development team has made upgrading an existing CP/M or ZRDOS system a snap—menu-driven installation and configuration utilities do all the work. And the documentation is top-notch.

I'm enthusiastic! ZSDOS boosts CP/M 2.2 computing to a new level of performance, increases reliability, and makes datestamping available to every Z80 computer. If you are a CP/M 2.2 or ZRDOS user, you will benefit most by upgrading to ZSDOS without delay. It's available from Plu*Perfect Systems.

BackGrounder ii Update

BackGrounder ii, as many readers of Jay Sage's column know, is a task-switching operating system system extension of CP/M 2.2, ZSDOS, and ZRDOS. Simply put, it allows you to switch back and forth between virtually any two applications programs, literally in mid-sentence! One reviewer described it as windows for CP/M, other users refer to it a super-Sidekick (it provides a calculator, notepad, screendump and background printing).

As TCJ rolls off the press I expect to have BackGrounder ii updated to full compatibility with ZCPR version 3.4. This will become the standard version, and currently licensed users can order an update from Plu*Perfect Systems.

File Systems

The main topic for this issue's Advanced CP/M column is file systems. Operating systems separate the organization and maintenance of a *file system* from the storage and retrieval of data on physical media. Files are most often stored on magnetic disks, and the portion of the CP/M operating system responsible for the file system is indeed called the basic *disk* operating system (BDOS).

In contrast, the lower-level tasks of actually writing data to, and reading them from, the physical disk is delegated to a *disk driver*, code that is part of the BIOS—the basic input/output system that must know the particulars of the specific hardware of the host computer.

The separation of file system functions and hardware-specific functions is fundamental to the design of any major operating system, and it has far-reaching implications.

First, it makes it possible to use the same programs on different computers, with different physical disk drives, provided that they run the same operating system.

Second, by keeping the logical organization of a "disk" and its physical realization in separate layers of the operating system, we can use a wide variety of storage media with the same file system. A ram "disk", after all, doesn't spin at 300 rpm, and a cassette tape or local area network is hardly a conventional disk, either. Yet, to a program and the BDOS, a file is a file is a file.

Third, with some extensions of the operating system, it is possible to mount a *different* file system on the same computer. For example, some FORTH operating systems run on top of

CP/M and provide access to both FORTH file screens and CP/M files. In a different way, DosDisk provides direct, transparent program access to MSDOS files in a CP/M environment.

Format Proliferation

The earliest CP/M computers had only a single format, the single-sided single-density 8" IBM "standard", and no provision for anything else. Then, as a few higher-performance and higher-capacity formats were introduced, they were hard-coded into the BIOS. Each new format required re-coding and reassembly of a new system.

Today, CP/M suffers from a surfeit of physical floppy disk formats. It seems that every manufacturer felt impelled to put his own label on yet another non-compatible format, to the point where we have well over 100 different ways of storing the same file on one 5¼" disk! This has also created something of an identity crisis, because it is not always possible to unambiguously determine a disk's format by magnetically reading the data on it.

The most modern BIOSes rise above this morass with flexibility and a degree of intelligence. They are able to identify a set of "native" formats, and automatically adapt themselves to the disk in each drive. In addition, they allow an external utility to set a drive to a "foreign" format, one that the BIOS cannot identify from its built-in data, but is known to the utility.

One such BIOS is the Advent TurboRom, written by Plu*Perfect Systems for the family of Kaypro computers. It automatically identifies 11 formats (Kaypro, Advent, Osborne, Ampro, Xerox, etc.). A companion program, MULTICPY (sold separately), allow TurboRom-equipped Kaypros to format disks in foreign formats, and to make exact copies of entire disks in those formats. And the TURBOSET utility allow the user to specify some 90 foreign formats, making nearly every 5¼" MFM-coded soft-sector format disk directly usable on a Kaypro computer.

MULTICPY and TURBOSET use a database (in dBase II format) of physical and logical disk formats. Because the database is extensible, new formats can be added. At Plu*Perfect Systems we use MULTICPY to produce distribution disks in many popular formats. If you have an unusual one, and can supply the physical and logical disk parameters and a sample disk, we can probably add it.

If your BIOS isn't this up-to-date, it is possible to temporarily replace its disk driver functions with a special application program long enough to copy files to or from a foreign format disk. Such a program must be written for your specific computer's hardware. Two popular utilities of this sort are UniForm (MicroSolutions) and Media Master (Intersecting Concepts).

You will find a cross-format tool is essential if you need to exchange data on a format not supported by your computer. The TURBOSET approach is the most flexible. It lets you use the foreign format disk with any regular CP/M program, just like your native-format disks. With the other tools you load the format-conversion utility, copy the needed file(s) to or from your native-format disk, remove the utility, and then run your regular programs.

There are a host of challenges that confront the programmer who seeks to upgrade his or her BIOS to this modern level of performance, and perhaps we can explore them in another column. In the remainder of this issue, however, we will have our hands full covering the file system and its implementation in the CP/M BDOS.

File Structure

Every file structure has two key properties—a method of naming files, and a method of allocating space for storing data.

Each file has a unique name within the filename space on the disk. (In CP/M, the filename space is a user number; in MSDOS and UNIX it is a subdirectory). With the name are usually a set of file attributes that may control permissions on access to the file, and perhaps datestamps as well.

Storage of data for the file is allocated in blocks—chunks of 128 or more bytes of data. With each filename the file system associates an ordered list of blocks, and a total length of the file.

The file system must maintain this information in an orderly fashion for each file on the disk. To do so, it uses a *directory* of filenames, a *free list* of unused data blocks, and an *allocated list* of blocks in use by the files.

The directory contains (at least) one entry for each filename. The entry will usually include permissions or attributes that control access to the file itself, and perhaps the datestamps for the file. And it will include some type of link to the file's data blocks.

The free list is some type of data structure that indicates which data blocks on the disk are not in use and can be allocated for writing data. On a fresh disk it will include all blocks of the disk not reserved for the directory, the boot code, or other operating system purposes. As a file is written, blocks are transferred from the free list to the allocated list and assigned to the file.

The Allocated List

I haven't said anything yet about how the directory and allocated list are actually stored. Those are key choices made by the designer of the operating system, and it's instructive to see how they can differ.

In MSDOS, the list of blocks is encoded in a file allocation table (FAT). The FAT has an entry for each data block (called a cluster in MSDOS) on the disk. An entry indicates that the block is unallocated (and is thus part of the free list), is allocated to a file, or is otherwise reserved.

The FAT is encoded in a way that allows it to serve two functions—it records the allocated and free blocks, and it shows which blocks are associated with which files. Blocks that are allocated to one file form a *linked list*. Each entry in the FAT is a pointer to the next block in that file's list, and the last entry is a special end-of-list mark.

The MSDOS directory entry includes only a pointer to the *first* block of the file. The rest of the blocks are obtained by following the linked list in the FAT. The FAT itself is stored on the disk, and the MSDOS system keeps a copy of it in working system memory. Thus, there are two separate data structures on an MSDOS disk—the FAT (which is actually stored in duplicate) and the directory.

CP/M takes a different approach—it includes the storage information as well as the filename information in the directory entry. Each directory entry contains a set of data block numbers and there is no file allocation table. To obtain the data blocks for a CP/M file, the system finds the first directory entry and reads off the block numbers.

Where is CP/M's free list? It is implicit in the directory. When a disk is logged in, the CP/M BDOS reads through the directory of a disk and keeps track of each data block that is allocated to a file. It encodes this information in an *allocation bitmap* for the disk, setting one bit for each block that is in use. The bits that are not set then represent the free blocks.

The UNIX system uses aspects of each approach. Each UNIX directory entry includes the filename and an *i-node* number; this is much like MS-DOS. An i-node is a list of the first 10 (512-byte) data blocks of a file, plus links to indirect lists of additional blocks. Directly including the list of the first 10 blocks in the i-node (a bit like including the block numbers in the first CP/M directory entry) allows UNIX to rapidly retrieve smaller files and yet use linked lists to extend files to very large sizes.

How Much Space Left?

One perennial disaster with many early CP/M programs, famous and obscure alike, was writing a file to an almost-full disk, running out of space during the operation, and having the program quit with the precious data lost forever. Of course, a well-written program wouldn't quit when a BDOS error occurs; it would clean up its incomplete file, allow the user to change disks,

reset the disk system, and re-write the file.

But a really well-crafted program wouldn't even attempt to write to the almost-full disk. Instead, before writing, it would determine whether there is enough space left on the disk to hold the file.

To do this, the program must obtain the total number of free blocks. This is a natural function for the disk operating system to perform, and in CP/M Plus there is a system call for this purpose (46). But it wouldn't fit into the space on the system tracks of the original 8" CP/M 2.2 systems, and so the BDOS includes another system call (27) to return the address of the drive's bitmap, and programs must count up the free blocks themselves.

Figure 1 show the Z80 routine, get_freek, that returns the number of unallocated kilobytes of space on the currently logged drive. It is portable—it works under CP/M 2.2, CP/M Plus and even for an MS-DOS disk when running DosDisk. The code includes contributions from Jay Sage, Joe Wright, and others, and is used, in a slightly varied form, in the SP (space) command in Z3PLUS and NZ-COM.

The routine first determines which version of CP/M is running. If the system is CP/M Plus, the BDOS will do all of the work. In fact, it's necessary to let it do the work, because in most CP/M Plus systems the allocation bitmap will be stored in a different memory bank and therefore not readily accessible to the program. (If the routine did attempt to use the bitmap address, it would add up bits of whatever program or data happen to be in that part of the main memory, resulting in an incorrect value).

CP/M Plus function 46 returns the space remaining on the disk as a 24-bit number in the first three bytes of the dma, in units of 128-byte records. So, to use this function, get_freek first sets the dma address to the temporary buffer at 80h and calls function 46. The divide-by-8 code then converts this to kilobyte units.

If the routine is running under CP/M 2.2, it first calls function 31 to get several disk parameters for the logged-in drive—the block-shift factor, the extent mask, and the maximum number of blocks on the drive. Next, it calls function 27 to get the address of the bitmap (allocation vector). The code at label "cntfree" then counts the number of unset bits in the bitmap, accumulating the count in register DE.

Since each block represents some multiple of 1K (1024 = 2**10 bytes), the code at label "free2k" multiplies the free block count by the size of one data block. The block shift factor is the base-2 logarithm of the number of 128-byte records per data block. In other words, it is the exponent in this equation:

block size in records = 2 ** block-shift-factor

If the block size is 1K (8 records), the block shift factor is 3 (i.e., 8 = 2**3), and the number of free blocks is already in 1K units. Otherwise, we multiply by the number of K in one block; this calculation is simply a 16-bit left shift that results from doubling HL (blkshf-3) times.

A Closer Look at the CP/M File Structure

One CP/M directory entry contains the following components:

user number—a logical partition of the volume (disk)
file name
file attributes
directory entry number
size of (the portion of) file indexed by this entry
the data block numbers for this entry

A single directory entry can hold either 16 8-bit data block numbers, or 8 16-bit directory numbers. A CP/M data block can be 1K, 2K, 4K, or 16K bytes (the blocking factor is part of the disk format specification), and the large blocks require 16-bit numbers. So a single directory entry may refer to a maximum of from 16*1K to 8*16K = 128K bytes of data, depending on the blocking factor for the disk.

Clearly, a file might be larger than the number of bytes that can be recorded in a single directory entry. To handle this case, CP/M creates *additional* directory entries to hold additional data block numbers. These entries have the same filename, user number and attributes as the initial entry, but they have unique directory entry numbers. (Contrast this with MS-DOS, which has just one directory entry, but a longer linked list of FAT clusters for a large file.)

Reading a file.

The actual numbering of CP/M directory entries is somewhat tortuous, and so we will discuss it later. First, let's get a grip on the details. Assume we already have a large file and consider first what the operating system does when an application program is reading the file.

First, the program calls the BDOS to open the file named in the indicated file control block (fcb). The CP/M BDOS searches for the initial directory entry, finds it, and stores the entry data, including the data block numbers, in the user's fcb.

Next, the program repeatedly calls the BDOS to read the file sequentially from the beginning. The (CP/M 2.2) BDOS gets the first data block number from the fcb, converts that value to track and sector numbers, and calls the BIOS to read one 128-byte record. Next, it increments the sector number (adjusting for reaching the end of a track) and calls the BIOS again, repeating for the number of records in a data block (8 in a 1K block, etc.). It then gets the second data block number from the fcb, converts to track/sector, and reads another set of records.

Eventually (after processing 8 or 16 blocks) all of the first directory entry's data blocks have been used, and the BDOS must search for and read in the next directory entry. (At this point, on a physical disk the movement of the disk heads back to the directory track can often be heard; this extra motion significantly slows down access to large CP/M files.) The BDOS then repeats the process of computing track/sector numbers and calling the BIOS to read records.

Writing a File.

Writing a file involves reversing these steps, with a few key additions, because disk space must be allocated. Let's assume our program is writing a new file.

First, the program calls the BDOS to create the file with the name stored in the fcb. The BDOS searches the directory for an empty (unused) directory entry. It then writes the new filename into that entry, with zeros for block numbers.

Now consider what the BDOS must do as the program sequentially writes the file. First, the BDOS must find a free data block on the disk. To do this it consults its free list for the disk (the allocation bit map) and assigns one block to the new file. It marks that block as used and puts the block number into the file control block. Now that the block number known, the next steps are much like reading—the BDOS translates from block number to track/sector numbers and calls the BIOS to write 128-byte records, until a block is full. Then, when a new block is needed, the BDOS gets the next free block from the free list, and repeats the process.

Eventually, the file control block is filled up with 8 or 16 data block numbers, and the BDOS must make a second directory entry. But before doing so, it "closes" the initial entry by writing the file control block values to that directory entry on the disk. Then, it searches for another empty entry, creates the second directory entry for the file (with the same name, but a different entry number), and finally resumes the process of allocating a data block and writing records.

At last, when the entire file has been written, the program calls the BDOS to close the file. Just as it did for the "internal" close of the initial directory entry, the BDOS writes the data block numbers in the file control block to the final directory entry on disk.

Figure 1. Free Space on a Disk

```
bdos    equ     5
tbuff   equ     0080h


; Enter:        a  = drive (0=A:, ..., 15=P:)
; Exit:         hl = free space on drive, in Kilobytes
;
get_freek:
        ld      (spacedrv),a    ; save drive
        ld      e,a
        ld      c,14            ; BDOS select disk function
        call    bdos
;
; check for CP/M Plus
;
        ld      c,12            ; get bdos version
        call    bdos            ; if not cp/m 3 system
        cp      30h
        jr      c,dparams       ; ..jump to calculate from alv
;
; calculate free space for CP/M Plus
;
        ld      de,tbuff        ; set default dma
        ld      c,26
        call    bdos
        ld      c,46            ; get disk freespace
        ld      a,(spacedrv)
        ld      e,a             ; ..on this drive
        call    bdos
;
; Disk space is returned by CPM+ at dma for 3 bytes.
;
        ld      hl,(tbuff)      ; Low to L, Mid to H
        ld      a,(tbuff+2)     ; High to A
        ld      b,3             ; Divide by 8 (SHR 3)
;
; Shift everything right into HL (64 MB max reportable)
;
div:    or      a               ; Clear carry
        rra                     ; High
        rr      h               ; Mid
        rr      l               ; Low
        djnz    div
        ret                     ; hl = space free in Kbytes

;
; For CP/M 2 use this method:
;
dparams:
        ld      c,31            ; BDOS get disk parameters function
        call    bdos
        inc     hl              ; point to block shift-factor byte
        inc     hl
        ld      a,(hl)          ; Get value and
        ld      (blkshf),a      ; ..save it
        inc     hl              ; point to max data block number
        inc     hl
        ld      a,(hl)
        ld      (extmsk),a      ; save it
        inc     hl
        ld      e,(hl)          ; Get (word) value into DE
        inc     hl
        ld      d,(hl)
        inc     de              ; Add 1 for max number of blocks

; Compute amount of free space left on disk

dfree:  ld      c,27            ; BDOS get allocation vector function
        push    de              ; Save BLKMAX value
        call    bdos            ; Get allocation vector into
        ld      b,h             ; ..BC
        ld      c,l
        pop     hl              ; Restore BLKMAX value to HL
        ld      de,0            ; Inititialize count of free blocks

; At this point we have
;       BC = allocation vector address
;       DE = free block count
;       HL = number of data blocks on disk

cntfree:
        push    bc              ; Save allocation map ptr
        ld      a,(bc)          ; Get bit pattern of allocation byte
        ld      b,8             ; Set to process 8 blocks
;
cnt2:   rla                     ; Rotate allocated block bit into carry flag
        jr      c,cnt3          ; If set (bit=1), block is allocated
        inc     de              ; If not set, block is not allocated, so
                                ; ..increment free block count
;
cnt3:   ld      c,a             ; Save remaining allocation bits in C
        dec     hl              ; Count down number of blocks on disk
        ld      a,l             ; if down to zero
        or      h
        jr      z,cnt4          ; ..branch
        ld      a,c             ; Get back current allocation bit pattern
        djnz    cnt2            ; Loop through 8 bits
        pop     bc              ; Get ptr to allocation vector
        inc     bc              ; Point to next allocation byte
        jr      cntfree         ; Process next allocation byte

cnt4:   pop     bc              ; clear stack
        ex      de,hl           ; Free block count to HL
;
```

```
        ld      a,(blkshf)      ; Get block shift factor
        sub     3               ; Convert to log2 of K per block
        ret     z               ; Done if 1K per block

; Convert for data blocks of more than 1K each

free2k: add     hl,hl
        dec     a
        jr      nz,free2k
        ret                     ; HL = amount of free space on disk in K
;
spacedrv:ds     1
blkshf: ds      1
extmsk: ds      1
```

Figure 2 . A CP/M Directory Entry

```
          + user number          +----EXtent byte
         /                      / +---S1 byte
        /                      / / +--S2 byte
       /     filename   type   / / / + record count
      / -------------- ----- / / / /
00    u f i l e n a m  e t y p x 1 2 r
10    - - - - - - - -  - - - - - - - -  data blocks
```

Figure 3. Calculate a Single Filesize

```
;
; Enter:        de -> fcb (36 bytes), freshly opened or
;                     copied from search-first buffer
;               extmsk contains extent mask for file's drive
;
; Exit:         a,hl = 24-bit file size value in 128-byte records
;
get_filesize:
        ld      hl,12           ; point to EXtent byte
        add     hl,de
        ld      a,(extmsk)      ; if not directory entry #0
        cpl
        and     (hl)
        jr      nz,g_rd         ; ..call bdos
        ld      b,(hl)          ; save logical extent #
        inc     hl              ; point to S2
        inc     hl
        ld      a,(hl)          ; or if overflow into S2
        and     7fh             ; (not directory entry #0)
        jr      nz,g_rd         ; ..call bdos
        inc     hl              ; or if Record Count
        ld      a,(hl)
        cp      80h             ; ..is full
        jr      z,g_rd          ; ..call bdos
;
; calculate filesize from fcb data
;
        ld      l,a             ; hl = rec. cnt. of last log. extent
        inc     b
        ld      de,80h          ; + 80h = size of each prior log. extent
        ld      h,d             ; h = 0
        jr      g_dj
g_lp:   add     hl,de
g_dj:   djnz    g_lp
        xor     a               ; clear high bits
        ret
;
; call bdos to calculate filesize
;
g_rd:   push    de              ; save fcb ptr
        ld      c,35            ; call bdos for filesize
        call    bdos
        pop     de
        ld      hl,33           ; point to random record #
        add     hl,de
        ld      e,(hl)          ; get it
        inc     hl
        ld      d,(hl)
        inc     hl
        ld      a,(hl)          ; high bits to A
        ex      de,hl           ; low 16 bits in HL
        ret
;-------------
```

If an error occurs during the process of writing the file, you may see some residue of the incomplete process. Quickie Quiz: Explain how each of the following might result:

1. Filename in directory, file is shown as 0K.
   (Editor: That is zero K, not Okay.)
2. Filename in directory, file is shown as 16K (or 32K or ...), but the end of the file is missing.

Internals of the Directory Entry.

Now we turn to the nitty-grity, and it is unavoidably confusing! It's also essential if you intend to really understand CP/M files.

The CP/M directory structure is like a tree house that grew as the kids got bigger. First it was a simple platform (for CP/M 1.4 files). Rooms got rebuilt to handle larger files and larger disks, and the file control block got extended to provide random access (CP/M 2.2). And small passageways were crammed with filesize, datestamps, and passwords (CP/M 3).

Some of the confusion is simply terminological. One directory entry is 32 bytes of data. Sometimes it is also called a physical directory extent—"physical" because it refers to actual bytes on the disk. Whenever you see this topic discussed, read carefully—I suggest you translate all references from "physical extents" to "directory entries", and reserve the term "extents" exclusively for "logical extents," which we will examine soon.

The directory entry has several fields, shown in Figure 2. The information is densly packed. You can look at an actual sector, which contains 4 directory entries, with the DU (or DU3) utility, or by running the following bit of code under a debugger and then displaying the default buffer at 0080h.

```
ld      c,11
ld      de,5C
ld      a,3F
ld      (de),a
call    5
rst     38
```

Byte 0 of a directory entry (labeled "u") is the file's user number. A value of E5 hex indicates that the entry is unused. Otherwise, it can have a value of 0 to 31 in CP/M 2.2. In CP/M Plus user numbers are restricted to 0 to 15, and higher numbers indicate special datestamp, password, and volume label entries.

Bytes 1-8 are the filename and bytes 9-11 the filetype. They must be uppercase, 7-bit letters, numbers, or a few other symbols. Each of the 11 high (eighth) bits of the filename and filetype are file attributes. Attributes 5-11 are reserved for the system to designate files are read-only, archived, and so forth.

The next four bytes encode the entry number and the length of the file. They will get our full attention in a moment.

Bytes 16-31 (10h-1Fh) are where the data block numbers are stored. These are either 16 1-byte values, or 8 2-byte values, depending on the disk format. If there are no more than 255 (FF hex) block numbers on a disk (for example, on a single-sided single density disk), it's possible to use 1-byte values. Otherwise, 2-byte values are needed.

The Directory Entry Number

Now, had the tree house been built in one day, the directory number would be a 16-bit word. Instead, we have to climb through some tangled vines. So, hold on!

The CP/M file system has two fundamental units of measurement:

1 record = 128 bytes
1 logical extent = 128 records = 16K bytes

Records and logical extents are numbered sequentially, beginning with 0.

Now consider a 17K file, with copies on several types of disks. Things might look like this:

On Disk #1, 16K of data blocks fill up one directory entry. Then one entry corresponds to one logical extent. The 17K file will have 2 logical extents, and 2 directory entries.

On Disk #2, 32K of data blocks fill up one directory entry. (How might this occur? Suppose a block is 4K, and block numbers are 2-byte values. 8*4K = 32K.) Now, one entry can hold two logical extents. The 17K file will have 2 logical extents, but only one directory entry.

CP/M keeps track of logical extents with the EXtent byte, which can hold 0 to 31 (0 to 1F hex). After 31, it must again be 0.

Why, you may well ask, does CP/M not allow more than 32 extent values in this field? Well, the tree house architect wasn't that farsighted. In the directory search functions, the BDOS uses a '?' character to indicate a "wild-card" search. When a '?' appears in the EXtent byte of an fcb, the BDOS will match any extent number. And since the '?' byte is 3F hex = 00111111 binary, only 5 bits are available to number logical extents!

If five bits were indeed all that is available, CP/M files would be restricted to a maximum size of 32*block size. To allow larger files, the tree house added the S2 byte. It holds the "overflow" from the EXtent byte. Each unit of S2 thus represents 32 logical extents, and the the S2 byte can take a value from 0 to 3F hex.

The full logical extent number is, therefore obtained by combining the EXtent byte and the S2 byte as follows:

```
log_ext = (EXT & 1Fh) + ((S2 & 3Fh) << 5)
```

(I use the C language operators: '&' is bitwise and, '<<' is shift-left).

Note well that the high-order bits must really be masked; while the directory entry is active in the fcb, the BDOS uses the higher bits of the EXTent and S2 bytes for internal BDOS flags.

Now, what is the directory entry number (the "physical extent")? It is the logical extent number, divided by the number of logical extents per directory entry. And that depends on the format, information that is *not* in the directory, but in the BIOS's data structure for the drive—the disk parameter block (dpb).

```
entry_no = log_ext / extents_per_entry
```

The *extent mask* byte in the dpb encodes the number of logical extents per directory entry. Its value is

```
extent_mask = 2 ** extents_per_entry -1
```

A strange, but handy, representation, because it gives the number of times to right-shift the log_ext value to calculate the directory entry number. And, simultaneously, it is a bitmask that, applied to the EXTent byte, yields the number of logical extents within the current directory entry that are in use.

```
entry_no = log_ext >> extent_mask
         = ((EXT & 1fh) >> extmask) + ((s2 & 1fh)
             << (5 -extmask))
```

Fast Filesize Computations.

How big is a file? What is its size in records, or equivalently, what is the record number of the file? It is the record count in the last directory entry (the number of records in the final logical extent), plus the size, in records, of all prior extents. Since the RC byte may be 80 hex, we must mask it. The formula is:

```
recno = log_ext << 7 + (RC & 7Fh)
```

Before considering practical answers to that question, let's consider how large a record number can ever be. The record count is 7 bits, the EXtent byte is 5 bits, and the S2 byte can be 6 bits, a total of 18 bits. The largest possible record number is therefore 2**18. Since there are 8 = 2*3 records in 1 kilobyte, the maximum filesize is 2**15 K = 32 MB, a large file indeed!

This is the limit under CP/M Plus and ZSDOS. Regular CP/M 2.2, however, limited the record number to a 16-bit quantity (with the largest S2 value being 0F hex), and thus a maximum filesize of 4 MB. And I'm afraid most CP/M application programs expect that limit not to be exceeded.

We can determine a file's size in several ways. BDOS function 35 will return the filesize in the random record number field of the fcb. This is the easiest method; the BDOS does all of the tedious arithmetic, and the random record number field is 3 bytes, so it will hold a full 18-bit record number, should we ever have a file so huge. But it's slow, because the BDOS must search the directory from the beginning each time it is called.

A second method is to have the program read the complete directory, storing the directory entries for the file as it goes, and then find the last one. This is no faster for a single file, but it is a clear winner if the program is reading the complete directory anyway (in order to display it, for example). In this case, the file size calculation is made after the entries are stored and sorted by entry number (as well as alphabetically, perhaps).

A Single File's Size

Often enough, a program needs a file's size as an adjunct to other file operations. In this situation, the file can first be opened, or searched-for, and then its size quickly computed from the directory entry data. Figure 3 shows the routine, get_filesize, to perform this service.

If the file has only one directory entry, all of the information needed to calculate its size in records is available in the EXtent, S2, and RecordCount bytes returned in the fcb by an open call, or in the dma buffer by a search-first call. The routine first checks that that the fcb information is, indeed, for entry number 0. It then determines that there are no others by checking the record count, because if it is 80h (128), the entry is full, and there may be another one.

If all of these tests get passed, it calculates:

```
records = RecordCount + 128 * number
          of prior logical extents
```

Otherwise, it calls the BDOS, which returns the number of records in the random-record number field of the fcb.

The get_filesize routine returns the filesize as a 3-byte value in the A, H, and L registers. Except for very large files, A will be zero, and the filesize can be used as the 16-bit value in the HL register pair.

A List of File Sizes

What if you need to get the sizes of several files? If your routine has a lot of memory available to hold a large list of directory entries you can process them in a single batch. But in some applications memory must be conserved. The routine might be just a small part of a large program that need memory for other functions. Or perhaps it is a component of a Z-System resident command processor that wants to keep the TPA intact for the next GO command.

The most basic directory routine looks like this:

```
        set fcb to a wildcard mask
        set dma to a buffer
        search-first
          if not found, quit
loop: if entry number is 0, display entry at offset in buffer
        search-next
        if found, loop
```

How can we add the fast filesize calculation to this routine? Here's the sketch of the approach I used in the DIRectory command built into BackGrounder ii , and also later in JetFind. That command must be able to run when a regular program has been suspended, without molesting that program's memory. This is the special challenge.

We plan to modify the "loop" line to be:

```
if directory-entry is not full, calculate
                filesize from entry.
else use BDOS function 35.
```

Hmmm. Initially, this looks like it would be OK. In fact, we're in trouble as soon as it's necessary to use the BDOS filesize function, because that call will change the BDOS's internal directory pointers and mess up the next search-next call. This requires some discussion.

The BDOS search-first/search-next functions are unlike any other file functions, in that they are logically a single function that is called repeatedly at two entry points. This operation says, in effect: Find the first entry in the directory matching the supplied fcb and return it in the dma buffer. Thereafter, when entered at the search-next point, continue the search for the next matching entry.

The BDOS uses internal pointers to keep track of both the fcb and where it is in the directory search, and it presumes that there will be no intervening file operations except more search-next calls.

But, with some cleverness, we can get modify our routine further to get around this complication. After making the BDOS 35 call, we do a search-first call for entry 0 of that file. This resets the internal pointers to the spot where the previous search had last matched. Then, we search-next for the next entry.

The routine now looks like this:

```
    set fcb to a wildcard mask
    set dma to a buffer
    search-first
      if not found, quit
loop:  If directory-entry is not full, calculate
                filesize from entry.
    Else
        call BDOS function 35
        set fcb to last-found entry
        search-first
    search-next
        if found, loop
```

What's Next?

File systems are a big topic, we're out of space, and coding the little directory routine must be left as "an exercise for the reader."

I appreciate your comments and welcome suggestions for future columns. Topics I have in mind include stack and interrupt management and environmentally-aware programming. What else would you like to see? Drop me a line at Plu*Perfect! ■

REL-Style Assembly Language for CP/M and Z-System

Part 1: Choose Your Weapons

by Bruce Morgan

The virtues of assembly language programming for limited-memory environments like CP/M and its more flexible descendants, like ZCPR3x/Z-System and BackGrounder II, are undeniable. There is nearly unanimous agreement among users that nothing makes more efficient use of scarce RAM and disk space—or runs faster—than the tight machine code that only thoughtfully crafted assembler can produce.

For the casual programmer/hobbyist whose main exposure to computer languages has been MBASIC or perhaps Turbo-Pascal, coding entire projects in assembly language tends to be an intimidating prospect. Available instruction books generally date from the early 1980s and are usually based on the relatively primitive assembly language tools bundled with CP/M 2.2: the ASM assembler, LOAD hex-to-comfile utility, and DDT debugger. These programs work in their limited ways, but demand a great deal of the programmer and suffer significantly in productivity comparisons with high level languages. The results can be gratifying (such well respected Public Domain contributors as Irv Hoff still work in what amounts to ASM-style assembler), but the programmer-hours involved often fill up one's entire quota of free time in very short order.

Through dropping ASM and company in favor of more advanced tools and techniques, assembly language can undergo a breathtaking transformation, acquiring much of the assumed productivity advantage of Pascal and "C" while retaining its size and speed edge over all high level languages. All that is needed initially are a REL-capable assembler and linker along with the free (although copyrighted) subroutine library by Richard Conn called SYSLIB—Z-System/BGii users will also want Conn's VLIB and Z3LIB.

Choosing the Assembler and Linker

The first thing to consider in choosing an advanced CP/M-compatible assembler is whether to stick with ASM-style Intel 8080 mnemonics or to make the leap over to the "official" Zilog presentation of the Z80 instruction set. I much prefer Zilog, but if you've programmed with ASM or MAC for a while, sticking with Intel mnemonics is not all that sinful and can make your learning curve a bit less steep.

The single essential characteristic required of the assembler is the ability to produce object files in Microsoft relocatable (filename.REL) format. If you run CP/M 3.x or, perhaps, MP/M 2.x, you probably own such an assembler already, Digital Research's RMAC. If you've made the decision to stick with Intel mnemonics for now, RMAC can be your assembler for now. It's a proven, reliable tool, though a bit short on features and, like its sibling, MAC, notoriously slow.

RMAC's companion linker, LINK, is an exceptionally fine and robust tool. Only the nearly $200 SLRNK+ matches it in output versatility (only these two can produce the exotic-but-useful PRL and SPR formats) and it handles segment sequencing (not crucial for beginners, but very valuable later on) flawlessly. Its only major defect is its speed, so color LINK slow-but-steady. We got a copy of LINK with a used computer that has long since bit the dust, but the tool has a secure place in our hearts and on our hard disk. The consistent correctness of its output make it our standard by which other linkers are evaluated.

Microsoft's classic M80 assembler is on the next rung of assembler capability. In its default 8080 mode, M80 performs like an enhanced, somewhat more fault-tolerant RMAC, but it has another capability that makes it near-indispensable to the serious assembler jockey: it also understands Zilog mnemonics, making it the only known tool to handle both major dialects of CP/M-compatible assembly language. It's available cheaply (about $25-$40, bundled with Microsoft's FORTRAN or BASCOM packages, from surplus outlets and at flea markets) and represents the closest thing to a "de-facto standard" assembler we have. Almost any assembly language source file in the Public Domain can be handled nicely by M80, with little or no modification.

On the down side, M80 isn't much faster than RMAC and its companion linker, L80, isn't anywhere near as nice a tool as it could (and should) have been—use LINK, PROLINK or an SLR linker (described below) instead.

The assembly language tools of SLR Systems are the epitome of speedy, modern assemblers and linkers. They come in a number of "flavors": SLR-MAC replaces MAC, RMAC and the 8080 mode of M80 (with the Z80 extensions of Z80.LIB built-in), Z80ASM handles code written for M80's Zilog mode, and SLR180 is essentially Z80ASM with Hitachi HD64180 opcodes added.

All three are available in standard and "virtual" versions, with the standard editions being very economical (about $50) and the pricier ($195) virtual tools

Bruce Morgen is the founder and "Director" (in that he directs himself and occasionally attempts to direct his wife, Julie) of the North American One-Eighty Group (NAOG), a band of about 400 hardy computing enthusiasts all over the world who use HD64180-based computers and/or Z-System compatible operating systems.

A former Associate Editor of Electronic Products magazine and "Advanced User" columnist for User's Guide, Bruce has been a freelance writer/editor and programmer since 1985 but will consider any and all alternatives involving great wealth. He can usually be reached via voice phone at 215-443-9031, hardcopy mail at N.A.O.G., P.O. Box #2781, Warminster, PA 18974, U.S.A., or by modem at Lillipute Z-Node (312-649-1730) and Drexel Hill NorthStar (215-623-4040), 300/1200/2400 bps, 24 hours.

adding such Rolls-Royce features as listing missed relative jump opportunities and handling very large source and listing files.

The SLR linker also comes in two versions. SLRNK is what L80 should have been in the first place, while SLRNK+ isn't memory-bound and adds the key feature of automatically putting code, data and common segments in predesignated order (more on that in Part III of this series). Why should you choose SLR over the older tools? That's a simple answer: S-P-E-E-D. They are simply incomparable in this respect, often completing their work in less than a quarter of the time required by the Digital Research or Microsoft products. They deserve your serious consideration if you place a high value on your time.

Free Alternatives

There is no truly legitimate Public Domain REL-capable assembler. Cromemco's Zilog-convention ASMB assembler (from its almost-CP/M CDOS operating system) found its way into the P.D. a few years back as "ZASM.LBR" and may still be floating around in your disk collection. If so, Cromemco reportedly says it's OK to use it, but the company discourages further distribution.

ASMB/ZASM is a sound tool with a number of advanced features and two major deficits. First of all, it is very slow, perhaps even slower than RMAC. Second, it requires a "virgin" operating system to avoid outright crashes—you must not be running any memory resident extensions (this even includes, regrettably, BGii) for it to work. That aside, ASMB/ZASM produces perfect REL output and nicely formatted listings—it even has the relative jump opportunity listing feature that SLR owners must pay $145 extra for! ASMB/ZASM's command line syntax pretty much mimics ASM's.

Free linkers are another and happier story. There are two good ones, PDLN and PROLINK. PDLN (distributed as PDLN10.LBR) is the work of Wilson Bent. It is rather slow and has a distinctly "UNIXy" command syntax, but these are not significant problems. It does segment sequencing sensibly, but its CRT messages are distinctly uninformative with regard to segment addresses—again, not a big problem and certainly nothing for beginners to worry over.

PROLINK (look for PROLINK.LBR), from Ron Fowler's NightOwl Software is a much more full-featured linker than PDLN and is one beautifully designed piece of software. PROLINK can function as a true, fully interactive linkage editor and it has a built-in command file capability and dozens of other features that even SLRNK+ can't match. It is also very fast and its (mostly) English command verbs make it very easy to learn and use.

As a matter of fact, PROLINK would be our linker of choice if it weren't for one major deficiency: PROLINK is a one-pass tool and cannot handle segment sequencing at all. Ron has informed us that all the "hooks" for proper two-pass operation are in place, but lack of demand and NightOwl's current emphasis on the IBM-PC version of his MEX communications program have precluded any further PROLINK development. Even with this serious flaw, PROLINK is a tool that every aspiring or experienced assembly language user should have on hand. It is a great program and we use it wherever possible.

Debuggers

There's nothing to stop you from using DDT or a similar non-symbolic debugger, although something fully Z80-compatible like C.B. Falconer's DDTZ is a better choice for stepping through REL-style assembly language programs. However, a symbolic debugger (one that can refer to label names rather than only absolute addresses) is often preferable.

The two industry workhorses for this job are Digital Research's SID (Symbolic Instruction Debugger) and ZSID (Zilog Symbolic Instruction Debugger). Both are durable, proven tools, with ZSID preferable because it can handle the entire Z80 instruction set properly. SID is part of the CP/M 3.x and MP/M 2.x distributions, unfortunately ZSID is not—and it's hard to find through surplus and flea market channels.

Luckily, there are two serviceable alternatives available in the Public Domain. Z8E (look for Z8E13.LBR) is the more spectacular of the two, an "animated" debugger released by author Rick Surwilo in 1985 that must be installed for your terminal if your CRT doesn't understand ADM3A-style cursor addressing. Quite a bit larger than DDT or ZSID, it is nonetheless a favorite of skilled assembly language programmers around the world. Somewhat plainer and more ZSID-like, Thomas Wagner's WADE (look for WADE.LBR) was released in mid-1988 and is also quite competently written and eminently usable.

By all accounts, the premier Z80 symbolic debugger is DSD, a $130 commercial program that was available from Echelon before that company's recent and unfortunate passing from the scene. My very trustworthy friend Jay Sage swears by it, but I'm not sure if it's still readily available.

Onward

Now that you're appropriately armed with an assembler, linker, and debugger, get ready to break out your favorite ASCII editor, because next time out we're going to explain the essential source file conventions and build a couple of REL-style programs together. ■

The ZCPR3 Corner

by Jay Sage

Jay Sage has been an avid ZCPR proponent since version 1, and 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 nearly five years and can be reached there electronically at 617-965-7259 (on PC-Pursuit) or in person at 617-965-3552 or 1435 Centre St., Newton, MA 02159.

Jay is best known for his ARUNZ alias processor, the ZFILER file maintenance shell, and the latest versions 3.3 and 3.4 of ZCPR. He has also played an important role in the architectural design of a number of programs, including NZ-COM and Z3PLUS, the new automatic, universal, dynamic versions of Z-System.

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.*

My column about shells and WordStar® Release 4 (WS4) in TCJ issue #33 prompted more than the usual level of commentary. There were extensive discussions on Z-Node-Central and the Lillipute Z-Node (the official TCJ bulletin board), and several messages reached me over the ARPA network. Not all of the comments were favorable, but I was nevertheless happy to receive them. They helped further clarify my thinking on the very important subject of shells and have spurred me on to prove my points by *actually converting WS4 to a ZCPR2-style shell*! After a bit of follow-up discussion, I will describe how this conversion was accomplished.

Corrections

There were some things I said in the previous column that were factually wrong, and before I do anything else I wish to correct them.

First, I stated that the Z-System code in WS4 was written by someone other than MicroPro. I was wrong. David McCord, who was vice president at Echelon at the time WS4 was developed, sent me a message with the facts of this matter. Echelon, through staff like David and through published materials, educated Peter Mireau of MicroPro on the facilities and philosophy of Z-System. Peter did all the actual programming, so the coding mistakes were his fault, not Echelon's or David's.

From a broader perspective, however, as I stated in the previous column, the real culprit was inadequate testing. Bugs in the code would have been discovered and conceptual issues clarified had more people in the Z community been involved as beta testers. There are so many different styles of using Z-System that it takes a number of testers to uncover problems. Within days after copies of WS4 were delivered to users of my Z-Node, I started getting questions about strange behavior exhibited by WS4, behavior that turned out to result from its operation as a shell.

A second mistake in the earlier column was my implication that WS4 does not get its own name from the ZCPR3 external file control block (XFCB). I no longer remember what made me think that this was the case, but David McCord assured me (and I have now verified for myself) that WS4 does, indeed, get its name from the XFCB when it sets up the shell stack entry.

Finally, one reader reported to me that my WSSHLOFF routine (the one that completely disables shells while WS4 is running and reenables them when WS4 terminated) crashed his system. Unfortunately, a large number of misprints crept into the listings in going from my disk file to the printed pages. Most of the typos were obvious, but one was compounded by a double error. In the WSSHLOFF listing, the value for EXITSUB was printed as 03BVh. The 'V' was obviously a mistake, and clever readers looked at the similar listing for WSSHLFIX, where the value was given as 03B3h. This, regrettably, looks correct but was also a typo. The proper value is 03BEh.

More WS4 Comments

While on the subject of WS4, I would like to add a few further comments about how it works. Not surprisingly (considering when it was developed), in creating its shell stack entry WS4 does not make use of the facility introduced with ZCPR version 3.3 that allows a program to determine from the XFCB not only its name but also the directory from which it was actually loaded (the user number is at offset 13 and the drive, with A = 1, at offset 14).

As a result, in order for WS4 to be reinvoked as a shell, the command search path must include the directory in which WS4 is located. I mention this here as a reminder and suggestion to authors of new or updated shells and error handlers that they use this Z33 facility to avoid the requirement that the program be on the path and to speed up loading of the program (by eliminating any search for it). My WordStar conversion described later adds this feature.

With WS4 it is generally necessary that the command search path include WS4's directory for an additional reason. I learned the hard way that when WS4 runs under Z-System, it pays no attention to the drive and user number that WSCHANGE specified as the location for the overlay files; it only uses the search path to try to locate them.

This is a problem for me because, as I have explained at length in previous columns, I put only my small RAM disk on the path and use ARUNZ aliases to invoke all programs except the very few that fit on the RAM disk. With this approach, there is no way to get WS4 to find its overlay files. The conversion addresses this problem also.

ZCPR2 vs. ZCPR3 Shells

I would now like to take up again one of the subjects raised in issue #33: ZC-

PR2-style versus ZCPR3-style shells.

First an aside. Shells seem to be a surprisingly emotional issue. I thought my earlier column presented a fairly carefully and calmly reasoned discussion of some aspects of shells, including their pros and their cons. Some readers, however, took great offense at my even questioning the current method of implementing shells or of what some people are trying to do with them.

One reader went so far as to suggest that I had no business commenting on the subject when, by my own admission, there are a number of shells that I have never used. Besides the fact that this is hardly a reasoned argument, I would like to make sure that the following facts about shells are fully appreciated.

Editor's Note: It would be nice if readers would forward a copy of of their comments to TCJ so that we could share them with the rest of the readers.

ZCPR3-style shells are a **facility of the command processor**. Without special code in the CPR, **there would be no such shells**. As the author of the two latest versions of the ZCPR command processor, I think I can speak with some authority (though certainly not with infallibility) on the subject, since in writing that code I had to consider the issue of shells rather carefully from a rigorous theoretical viewpoint.

ZCPR2-style shells, quite the contrary, are not a facility of the command processor; **they are a facility of the individual shell programs**. Their functioning depends only on the operation of the multiple command line facility. The command processor does not treat a Z2 shell command any differently than it treats any other command. This is really the key to the difference between the two shell implementations.

In the previous column I stated: "...I am coming to the conclusion that only programs like history shells...should be implemented as ZCPR3-style shells. Other programs, like ZFILER and WordStar should use the ZCPR2 style." I then invited readers to enlighten me if I was missing some important point. I got some responses to this invitation, but no one yet has offered me any evidence that I had missed any important point.

One reader reiterated essentially the same difference between Z2 and Z3 shells that I attempted to demonstrate with my example in which WordStar was invoked in a multiple command line. Apparently the point bears repeating.

This reader presented the point using a command line like the following:

```
ZFILER;ECHO TESTING
```

Under ZCPR2, ZFILER would run and present its file display to the user. If the user generated a command line "CMDLINE" as the result of a macro or in response to the prompt after the 'Z' command, a Z2-shell version of ZFILER would build the command sequence "CMDLINE;ZFILER" and insert it into the multiple command line buffer just before the next command to be executed. This would give:

```
CMDLINE;ZFILER;ECHO TESTING
```

The user's command line would run, and then ZFILER would be invoked again. Only on termination of ZFILER would the last command, "ECHO TESTING", be performed.

A Z3 shell would respond to the same command line from the user in quite a different way. As before, ZFILER would be invoked first. It would determine from the Z3 message buffer that it had been invoked manually and would respond by pushing its own name onto the shell stack. Then it would terminate. The command processor would then proceed to run "ECHO TESTING". Only after that, once the command line was empty, would ZFILER be reloaded, this time as a shell. Recognizing its shell status, it would now display its screen of file names and do its real work.

The reader who submitted this example, if I understood him correctly, viewed the Z3 behavior as correct and the Z2 behavior as wrong. If you are an experienced Z-System user, you will probably recognize in this reader a fellow expert (and, indeed, he is). He is so used to ZCPR3 that he no longer notices that it is the behavior of the Z3 shell that is truly bizarre!

Consider the following two command lines:

```
(1) ZFILER;ECHO TESTING
(2) ECHO TESTING;ZFILER
```

We have already analyzed the first one; the second one can safely be left as an exercise for the reader. We will simply state the answer that under ZCPR3 they will accomplish exactly the same thing! This is hardly a result that conforms to intuition, and I still remember in my early days as a Z-Node sysop trying to explain to quite a few users why the second command on a VFILER command line executes first!

Under ZCPR2 the result is just what one would expect. In the first case, ZFILER runs first, and ECHO runs only after the user terminates ZFILER using its 'X' command. In the second case, ECHO runs first and ZFILER second. In other words, with Z2 shells, commands are executed in the order they are entered, a notion that does not require long experience and great expertise to understand and get used to! And it gives the user a greater measure of control.

Mixed Z2 and Z3 Shells

The same reader submitted another interesting example that illustrates the confusing behavior that can arise when Z2 and Z3 shells are mixed. Here we assume that WordStar has been implemented as a Z2 shell and ZFILER as a Z3 shell. Suppose we use the 'R' command of WordStar to enter the command "ZFILER". WS4, as a Z2 shell, would generate the command line

```
ZFILER;WS
```

ZFILER, as a Z3 shell, would install itself on the shell stack and proceed to execute "WS". ZFILER would not run in its file maintenance mode until after we terminated WordStar.

This is, admittedly, probably not what one intended, since we most likely entered the ZFILER command with the intention of doing some file maintenance before returning to WordStar. On the other hand, it is certainly no more bizarre than what we saw in our earlier example.

If both WS4 and ZFILER were Z3 shells, then the invocation of ZFILER from the WS4 'R' command would cause it to become the active shell (the one on the top of the shell stack). The WS4 shell would be pushed down in the shell stack, and ZFILER would take control. With a little thought, however, you will see that the same is also true if both ZFILER and WS4 are Z2 shells!

The strange behavior with the mixed shells in the above example arises in part because ZFILER was not really being used as a shell in the Z3 sense, namely, as a replacement for the command processor's command-line input routine. It was intended as a file maintenance utility.

Suppose we had entered the command "EASE" (the Z3-type history shell) instead of "ZFILER" from our Z2 version of WordStar. This would establish EASE as the current shell and return to WordStar. That behavior would not seem strange, because in this case we would be thinking of our EASE command as establishing the shell to be used in place of the command processor the next time the command processor needed a new command line. So long as WordStar is running, there is no need for EASE to do anything. We expect it to take effect only after we are finished using WordStar.

Nested Z2 Shells and Recursive Aliases

Although I had once thought that the Z3 shell stack was required in order to nest shells, I showed in the earlier column that this is not the case. Z2-style shells can, in fact, be nested more flexibly. There is no predetermined limit to the

nesting depth or to the amount of information that can be passed with each shell command line. The only limit is imposed by the length of the multiple command line buffer, just as with the nesting of aliases.

With the standard shell stack configuration of 4 32-byte entries, if a shell command uses only 16 bytes, 16 bytes are wasted. On the other hand, if a shell command needs 48 bytes to hold its information, it cannot run at all under this configuration (NZ-COM can come to the rescue by allowing the shell stack configuration to be changed on the fly). With Z2 shells, these problems go away. In 64 bytes of command line, one can have two 32-byte shell commands or a combination of one 16-byte shell command and one 48-byte shell command (or five 12-byte shell commands).

I did overlook one point when I described putting data for the shells on the command line. In the Z3 shell stack, one can include, after the shell command's terminating null, any binary data that one wishes. Thus 256 values are possible for each extra byte in the shell stack entry.

In order to carry shell data on the command line, several additional constraints apply. First, the command processor strips the high bits off all characters in the command line, so only 128 values are available to start with. Secondly, the null character cannot be used because the command processor would interpret this as the end of the command line (that leaves 127 values). Finally, letters are converted to upper case, thereby making the characters from 'a' to 'z' inaccessible (scratch another 26). This leaves only 101 possible values out of the original 256. Moreover, extra characters are required as flags to signal the program to consider itself as having been invoked as a shell (a service provided in ZCPR3 by a flag in the message buffer). All of these things reduce the efficiency with which the space in the command line buffer can be used compared to the space in the shell stack.

One reader pointed out that recursive aliases cannot be used with Z2-type shells. This is true...but only if one is using the pseudo-recursive alias that I invented. This kind of alias accomplishes a crude approximation to recursion by discarding any pending commands in the command line buffer. This will, indeed, discard any shell reinvocation commands. However, if one uses the logically sound and rigorous recursive alias technique invented by Dreas Nielsen (see my column in issue #28), there is no problem. It sometimes pays to do things right!

In fact, it seems to me that the Z2 shell is, in essence, a recursive alias, a program that keeps invoking itself. And this is just what most (if not all) Z3 shells actually do. I am still awaiting an example of something (good) that a Z3 shell can do that cannot be done in some equivalent way with a Z2 shell or recursive alias.

The Real Difference Between Z2 and Z3 Shells

After much reflection, I think I have finally put my finger on the fundamental distinction between Z2 and Z3 shells. It derives from the facts I alluded to earlier: that the Z3 shell is a true creature of the command processor and the Z2 shell is not.

Here is an example that will illustrate the point. Suppose the history shell EASE were implemented as a Z2-style shell and that while it is running, we issue the command "DIR". EASE will insert into the command line a sequence like the following:

```
DIR;EASE
```

DIR will run, and then EASE will be reinvoked. Looks fine! But now suppose the user enters the command "IF EXIST FN.FT". EASE will then generate the command line

```
IF EXIST FN.FT;EASE
```

If the file FN.FT exists, this will again work just fine, but suppose the file does not exist. Then the system will enter a false flow state, and the EASE command (and perhaps other commands pending in the command line after it) will be flushed by the command processor. The shell function will be lost, and any other pending commands will be processed in an unintended way.

For a Z2 shell to function properly in general, all command lines inserted by it must result in the same flow state at the

end of the command line as at the beginning. With a MENU shell it could be possible for the system designer to guarantee this, since he can control which commands are generated by the shell. With a history type shell it would be nearly impossible to ensure that this condition would always be met.

The critical feature of shell processing under ZCPR3 is that **flow processing is suspended during the operation of shells.** This allows them to run, as they must, even after the user has passed a command that leaves the system in a false flow state. The *ZCPR33 Users Guide* goes into some detail on this matter, and had I remembered better what I wrote there, it would not have taken me this long to come to the essence of the Z2-vs.-Z3 shell issue.

Some users of ZCPR33 have modified the way the command processor deals with flow control in shell processing. No one has yet convinced me of the value of this (the risks are undeniable). It still seems to me that Z2-type shells and recursive aliases can accomplish the same thing, but in a logically sound way.

I have extended an invitation to Dreas Nielsen to write a series of columns for TCJ explaining his very powerful shell programs. Since he is also one of the people who has made this modification to the CPR, perhaps he will also present the other side of this story and explain why it is necessary or desirable to treat shells the way he does.

Remaking WordStar Release 4

When I first received my copy of WordStar 4 and encountered problems with the way it handled shells, I fired up the DSD debugger and tried to figure out how to fix it. After a considerable amount of rummaging about in the code (and especially trying to figure out what was going on inside WS.OVR), I gave up. Later I tried again...and failed again. In the course of preparing this column, I decided to have one more go at it, and this time things started to click.

The patches I will describe here are preliminary and have not yet been extensively tested. In fact, as I write this, I am the only one who has used them, and you know what I said above about the dangers of a test program that does not involve a variety of Z-System users. So, you are hereby recruited, if you are willing, to join the test program.

Since I may very well have made some mistakes, and since there are further changes that people may want to make (let's hear your suggestions), I will not only give the results; I will describe the process by which these patches have been developed.

```
E =  9h  the command search path (PATH)
E = 15h  the named directory register (NDR)
E = 18h  the multiple command line buffer (MCL)
E = 1Eh  the shell stack (SHL)
E = 22h  the message buffer (MSG)
E = 24h  the external file control block (XFCB)
```

Figure 1: Values of the E register.

Cracking the Code

The first step toward changing the code was figuring out how the virgin WordStar was doing what it did. In particular, I wanted to locate routines related to Z-System functions, so the first thing I tried was searching for all references to address 109h, which contains the address of the Z-System environment (ENV). Any WS4 feature that made use of a Z-System facility would have to get information from the ENV.

As best I recall, this did not turn up many references and did not particularly help (though it was a good idea, and that's why I mention it). In the end, I just started tracing the code from the beginning, figuring that WS4 would have to determine fairly early whether it was running under Z-System or standard CP/M. This turned out to be correct, and very soon I came to the key Z routine, at address 0AA4h in WS.COM. This routine returns the address and size of a Z-System module specified by an offset passed in the E register.

Having discovered this routine, I used DSD to find all references to it in WS.COM and WS.OVR. They occur with the values of E shown in Figure 1.

Setting up the Shell Stack

The block of code beginning around address 3CBFh in WS.OVR makes references to MCL, XFCB, and SHL. I guessed correctly that this had to be the code where WS4 sets up its shell stack entry. (This block of code, by the way, is where the shell-pushing mistake occurs for the case where the shell stack is currently empty.)

The patch for this part of WS.OVR (see Listing 1) modifies this code. First of all, since WS4 is going to operate as a Z2-type shell, we do not want it to do anything with the shell stack. It is easy to disable the code by simply skipping over it, but one has to watch out for subtleties. Indeed, in order for the 'R' command to use the MCL and not chain using the greatly inferior CP/M method, WS4 has to *think* that the shell entry was established successfully.

I noticed that a flag was being set into address 2200h, and I surmised that it is used by WS4 to show that it is running under Z-System. In the patch, I set this flag even though the shell stack entry is not being set up. I have not examined all references to this flag, and there is a chance that there are additional, more complex effects. If any problems appear with the patched version of WordStar, this flag might be involved. For the initial attempt at fixing WS4, I just took the easiest course of action, and so far it appears to have worked.

It seemed foolish to waste space in WS.OVR by doing nothing more than setting the flag and jumping to where the original code resumed (60AAh). Instead, I have used the space to compute the command line necessary to reinvoke WordStar. The code gets not only the name by which WordStar was invoked but also the drive and user number from which it was loaded. A command line of the form ";DUU:WSNAME" is generated.

There is one extra step in this part of the patch. When running as a Z3 shell, WS4 knows from the command status flag in the message buffer when it was invoked as a shell so it can put up the press-any-key message before clearing the screen and resuming operation. As a Z2 shell, WS4 cannot use this facility. Instead, a signal has to be included in the command tail. For reasons that I will not go into in full detail, I chose for this signal a comma at the very end of the tail. Very briefly, the comma is a handy character because it is not parsed into the default file control blocks, where a program could confuse it with a file name.

The final reinvocation command line, with its terminating null, takes the form

```
;DUU:WSNAME ,<0>
```

Since I could not be sure that this section of overlay code would persist in memory until the command would be used, I store it at the top of the WS4's user patch area (MORPAT).

LISTING 1

```
                ; Program:      WordStar Shell Modification Patches
                ; Author:       Jay Sage
                ; Date:         August 7, 1988
                ; Patches to make the WordStar Release 4 'R' command operate as a ZCPR2-type
                ; shell.  Several routines in WS.COM and WS.OVR must be changed.
                ; 1. WordStar must be prevented from pushing its name onto the Z-System shell
                ;    stack.  However, a flag that is used by the 'R' command to determine how
                ;    to operate must be set as if WS4 had set itself up as a shell.
                ; 2. The popping of the shell stack when WS4 terminates must be disabled.
                ; 3. The user input to the prompt from the 'R' command must be handled
                ;    differently.  A command to reinvoke WS4 must be appended to the user's
                ;    input, and then any commands pending in the multiple command line buffer
                ;    must be added as well.  The result is then placed into the command line
                ;    buffer.  If overflow occurs, the user command is ignored, and an error
                ;    message is displayed until a key is pressed.  The chaining command is of
                ;    the form '';DUU:WSNAME ,''.  The comma at the end of the command tail is
                ;    used as a signal that WS4 was invoked as a ZCPR2 shell.
                ; 4. An optional patch can be included to defeat the use of the path for
                ;    searching for the overlay files.  An internal path can be specified.
                ;---------------------------------------------------------
        0000    no      equ     0
        FFFF    yes     equ     not no
        FFFF    intpath equ     yes             ; Use internal path to find OVR files?
        5B04    morpat  equ     045bh           ; Patch area
        CB04    namebuf equ     morpat+128-16   ; Keep program ''DUU:PROGNAME ,<0>''
        A40A    envoff  equ     0aa4h           ; WordStar ENV offset routine
        0022    zflag   equ     2200h           ; Z-System running flag
        381F    rcmdbuf equ     1f38h           ; Buffer for 'R' command input
        391F    rcmd    equ     rcmdbuf+1       ; Beginning of user's command line
        8603    clrscr  equ     0386h           ; Clear screen character sequence
        C717    scrnfn  equ     17c7h           ; Routine to perform screen functions
        8002    conout  equ     0280h           ; Routine to output character in A to console
        0700    bell    equ     07              ; Bell character
                ;---------------------------------------------------------------------
                ;                       PATCHES TO WS.OVR
                ;---------------------------------------------------------------------
                ; Modifications to the code that pushes WordStar onto the shell stack.
                ; This patch prevents the WordStar shell entry from being set up, but it
                ; sets the flag in 2200h that makes WordStar think that it has set it up.
                ; In this way, the 'R' command will work as it would with shells engaged.
                ; The space is used to determine the command line needed to reinvoke
                ; WordStar.  The ZCPR33 facility for returning the directory in which the
                ; program was located is used to provide an explicit DU: prefix.  The
                ; resulting command line is kept at the end of the user patch area.
        8023    offset  defl    2380h           ; Real address = address in overlay + offset
        BF3C            org     3cbfh           ; Place to install patch
3CBF  1E 24             ld      e,24h           ; Get pointer to XFCB
3CC1  CD A40A           call    envoff          ; HL -> XFCB
3CC4  E5                push    hl
3CC5  11 0D00           ld      de,0dh          ; Offset to user number where WS.COM found
3CC8  19                add     hl,de
3CC9  46                ld      b,(hl)          ; User number to B
3CCA  23                inc     hl
3CCB  7E                ld      a,(hl)          ; Drive to A
3CCC  C6 40             add     a,'A'-1         ; Convert drive to letter
3CCE  21 CB04           ld      hl,namebuf      ; Point to buffer at end of patch area
3CD1  36 3B             ld      (hl),';'        ; Command separator
3CD3  23                inc     hl
3CD4  77                ld      (hl),a          ; Store drive letter
3CD5  23                inc     hl
3CD6  78                ld      a,b             ; Now work on user number
3CD7  0E 2F             ld      c,'0'-1         ; Tens value
3CD9            tens:
3CD9  0C                inc     c
3CDA  D6 0A             sub     10
3CDC  30 FB             jr      nc,tens
3CDE  C6 3A             add     10+'0'          ; Convert units to ASCII
3CE0  71                ld      (hl),c          ; Stash tens digit
3CE1  23                inc     hl
3CE2  77                ld      (hl),a          ; Stash units digit
3CE3  23                inc     hl
3CE4  36 3A             ld      (hl),':'        ; Insert colon
3CE6  23                inc     hl
3CE7  D1                pop     de              ; Get pointer to XFCB again
3CE8  06 08             ld      b,8             ; Maximum of 8 letters
3CEA            copyname:
3CEA  13                inc     de              ; Advance to next letter
3CEB  1A                ld      a,(de)
3CEC  FE 20             cp      ' '
3CEE  28 05             jr      z,copydone      ; Quit at first space
3CF0  77                ld      (hl),a
3CF1  23                inc     hl
3CF2  05                dec     b
3CF3  20 F5             jr      nz,copyname     ; Quit after eight characters
```

The Initialization and Termination Patches

Having made the above modification, we must make two others in order to remain consistent. First, we must modify the WS4 initialization code in which it determines whether or not it was invoked as a shell. This is the patch at address 1A2Fh in WS.COM. The patch calculates the address of the last character passed in the command tail and checks to see if it was a comma. If not, it proceeds with normal operation of the program.

If there is a comma, the shell-wait message must be displayed until the user presses a key. But one must also remove the comma from the command tail to ensure that WordStar not think it has been passed a file name. At present I do this by replacing the comma with a space. This is not rigorous, but it seems to work, since WS4 is apparently not confused by a tail consisting only of spaces (unfortunately, a number of programs are).

Since WS4 no longer pushes its name onto the shell stack, we must also prevent it from popping the shell stack when it terminates. This is the patch at address 13CEh in WS.COM. This is the easiest patch of all, since we simply have to skip some code. As an additional benefit, this frees up about 40 bytes of space that we use for some of our other patch code.

Fixing the 'R' Command

Now we come to the main item in this set of patches—the code that makes the 'R' command work as a ZCPR2-type shell. The new code here is much more complex than what it replaces, and we can only fit part of it at the original location 67B2h in WS.OVR. We put what we can there and continue with the rest in the MORPAT area in WS.COM.

The basic strategy is to take the command line entered by the user in response to the 'R' command prompt, append the WS reinvocation command (including its semicolon separator), and append any remaining command line pending in the multiple command line buffer (if there is one, it will begin with a semicolon also). If there is enough room for the result in the MCL, then it is moved there and chained to. If not, a warning message is displayed on the screen until a key is pressed, and the user command is ignored.

To implement this strategy, I chose the simplest method I could think of. Since the 'R' command operates from the WordStar no-file menu, the entire WS edit

buffer is available as a scratch area. I picked an arbitrary address of A000h for a buffer in which to build the new command line. Again, rigorous code calculate the address. My code is a quick-and-dirty solution.

Finding the Overlay Files

As I noted earlier, with my style of operation, WS4 had trouble finding its overlays. To solve that problem, the patch includes an optional section to install an internal search path for the overlay files. This patch is installed at address 0F5Fh in WS.COM, where it replaces a call for the location of the Z-System path with a call to a routine that returns the address and size of an internal path. In Listing 1 the internal path has the single element B4:, the directory in which I keep my WordStar program files. You can put any values you want here.

Installing the Patches

It is not possible to install the patches in WS.OVR using MLOAD or a debugger, because the OVR file is too large to load entirely into memory. ZPATCH, on the other hand, can handle the job splendidly. ZPATCH assumes an offset of 0000 for a file of type OVR, while the addresses in the listing are those shown when the file (as much as can fit) is read into memory under a debugger. To make things consistent, you should use the ZPATCH 'O' command to set the offset to 100.

Key in the new data carefully, checking from time to time that the address is still correct. Also, be careful not to go beyond a record boundary while in ZPATCH. It wraps from the end of the record back to the beginning of that record without warning (this really gave me grief until I caught on to the problem). When you get to the end of the current record, write it out (^W), advance to the next record (>), and reenter edit mode (E). Then you can resume entering data.

The attached listing was made with a specially configured version of the SLR Z80ASM assembler. Normally, I have it display addresses in logical order for easier interpretation. For hand keying of a patch, however, it is far more convenient to have the bytes of a word in *physical* order. Just watch out when reading the displayed symbol values. They, too, are stored in byte-reversed format.

```
3CF5                    copydone:
3CF5  36 20                     ld      (hl),' '        ; Put shell signal tail (a comma)
3CF7  23                        inc     hl
3CF8  36 2C                     ld      (hl),','
3CFA  23                        inc     hl
3CFB  36 00                     ld      (hl),0          ; Put in terminating null
3CFD  3E FF                     ld      a,0ffh          ; Fool WS into thinking shell installed
3CFF  32 0022                   ld      (zflag),a
3D02  C3 AA60                   jp      60aah
3D05                    endlpat:
        2A3D            endaddr defl    60aah - offset
        2500            free    defl    endaddr - endlpat
        0000                     if     $ gt endaddr
                                 endif
                        ;---------------------------------------------------------
                        ; This patch takes the user's response to the 'R' command, adds the command
                        ; to reinvoke WordStar, and appends any pending commands in the command line
                        ; buffer.  The result is written out to the command line buffer.  This
                        ; implements a ZCPR2-style shell for the 'R' command.
                        ; If the resulting command line is too long for the MCL, an error message is
                        ; displayed until a key is pressed, and then WS resumes as if no command line
                        ; had been entered.
                        ; The first part of this patch replaces code in WS.OVR.  There is not enough
                        ; space there for all the code, so it continues in the user patch area.
        00E2            offset  defl    -1e00h          ; Real address = address in overlay - offset
        00A0            scratch equ     0a000h          ; Area to use as scratch buffer
        B267                    org     67b2h           ; Address in WS.OVR
67B2  21 381F                   ld      hl,rcmdbuf      ; Point to 'R' command buffer
67B5  4E                        ld      c,(hl)          ; Get length into BC
67B6  06 00                     ld      b,0
67B8  23                        inc     hl              ; Point to user's command
67B9  11 00A0                   ld      de,scratch      ; Scratch buffer in RAM
67BC  ED B0                     ldir                    ; Copy user's command to buffer
67BE  21 CB04                   ld      hl,namebuf      ; Point to WS4 reinvocation command line
67C1  CD D113                   call    cpy2nul         ; Copy through ending null
67C4  C3 5B04                   jp      morpat          ; Continue in patch area
67C7                    end2pat:
        0400            free    defl    67cbh - end2pat
        0000                     if     $ gt 67cbh
                                 endif
                        ;-------------------------------------------------------------------
                        ;                       PATCHES TO WS.COM
                        ;-------------------------------------------------------------------
                        ; This is the continuation of the patch in WS.OVR that inserts the user's
                        ; command line, together with the WS reinvocation command, into the multiple
                        ; command buffer.
        5B04                    org     morpat
045B  D5                        push    de              ; Save pointer to buffer
045C  1E 18                     ld      e,18h           ; Get pending commands from MCL
045E  CD A40A                   call    envoff
0461  5E                        ld      e,(hl)          ; Get pointer to next command into DE
0462  23                        inc     hl
0463  56                        ld      d,(hl)
0464  EB                        ex      de,hl           ; Switch into HL as source for copy
0465  D1                        pop     de              ; Destination pointer into buffer
0466  CD D113                   call    cpy2nul         ; Copy through ending null
0469  21 8603                   ld      hl,clrscr       ; Clear the screen
046C  CD C717                   call    scrnfn
046F  1E 18                     ld      e,18h           ; Get MCL pointer
0471  CD A40A                   call    envoff          ; HL -> MCL buffer, A = max characters
                                ; Check length of new command
0474  11 00A0                   ld      de,scratch      ; Point to new command line
0477  47                        ld      b,a             ; Max length in B
0478                    lenloop:
0478  1A                        ld      a,(de)
0479  B7                        or      a
047A  28 11                     jr      z,oklength
047C  13                        inc     de
047D  10 F9                     djnz    lenloop
047F  11 9F04                   ld      de,errmsg       ; Display error message
0482  0E 09                     ld      c,9
0484  CD 0500                   call    0005h
0487  CD 491A                   call    sak             ; Wait for key to be pressed
048A  C3 4E7F                   jp      7f4eh           ; Pretend no user input
048D                    oklength:
048D  11 0400                   ld      de,4            ; Offset to command line in buffer
0490  EB                        ex      de,hl           ; Reverse pointers
0491  19                        add     hl,de
0492  EB                        ex      de,hl           ; HL = MCL, DE = MCL+4
0493  73                        ld      (hl),e          ; Set up pointer in MCL
0494  23                        inc     hl
0495  72                        ld      (hl),d
0496  21 00A0                   ld      hl,scratch      ; Source for command line
0499  CD D113                   call    cpy2nul         ; Copy it in
049C  C3 F613                   jp      13f6h           ; Chain to command line from WS
049F                    errmsg:
049F  07 4D 43 4C               db      bell,'MCL Ovfl - press any key...$'
```

```
04A3  20 4F 76 66
04A7  6C 20 2D 20
04AB  70 72 65 73
04AF  73 20 61 6E
04B3  79 20 6B 65
04B7  79 2E 2E 2E
04BB  24
04BC                    end3pat:
       0F00             free    defl    namebuf - end3pat
       0000                     if      $ gt namebuf
                                endif
                       ;-------------------------------------------------------------
                       ; This optional patch causes WS4 to use an internal path to locate
                       ; its overlay files.
       FFFF                     if      intpath
       5F0F                     org     0f5fh           ; This is where z3 path location is determined
0F5F  CD D913                   call    setpath         ; Call alternative routine
0F62  00                        nop                     ; Must fill 5 bytes
0F63  00                        nop
                                endif   ;intpath
                       ;-------------------------------------------------------------
                       ; Modification to the termination routine that pops the shell stack.
                       ; This patch eliminates the popping of the shell stack on exit from
                       ; WordStar.  The space from the end of this patch to 13f6h is available
                       ; for other uses (40 bytes).
       CE13                     org     13ceh
13CE  C3 F613                   jp      13f6h           ; Exit routine
                       ; This routine copies the string pointed to by HL to the address pointed to by
                       ; DE until a null byte is encountered.  The null byte is copied as well.
13D1                    cpy2nul:
13D1  7E                        ld      a,(hl)          ; Get source character
13D2  12                        ld      (de),a          ; Put into destination
13D3  B7                        or      a               ; Check for null
13D4  C8                        ret     z               ; If so, quit
13D5  23                        inc     hl              ; Bump up pointers
13D6  13                        inc     de
13D7  18 F8                     jr      cpy2nul
                       ; Alternative internal path routine
       FFFF                     if      intpath
13D9                    setpath:
13D9  21 E013                   ld      hl,path0        ; Point to internal path size
13DC  7E                        ld      a,(hl)
13DD  23                        inc     hl              ; Point to actual path
13DE  B7                        or      a               ; Set flags
13DF  C9                        ret
13E0  02                path0:  db      2               ; Allow up to two elements
13E1  02                        db      2               ; Drive (A=1)
13E2  04                        db      4               ; User
13E3  00 00                     db      0,0             ; Space for another entry
13E5  00                        db      0               ; Terminating null
                                endif   ;intpath
13E6                    end4pat:
       1000             free    defl    13f6h - end4pat
       0000                     if      $ gt 13f6h
                                endif
                       ;-------------------------------------------------------------
                       ; Modification to initialization code where WS4 determines if it was
                       ; invoked as a shell.  We have defined a convention where a comma on
                       ; the end of the command line signals WS4 to display its shell-wait
                       ; message and wait for the user to press a key.
       2F1A                     org     1a2fh
1A2F  21 8000                   ld      hl,80h          ; Point to command tail
1A32  6E                        ld      l,(hl)          ; Get length into L
1A33  CB FD                     set     7,l             ; In effect, add 80h
1A35  7E                        ld      a,(hl)          ; Get last character
1A36  FE 2C                     cp      ','             ; See if it is a comma
1A38  20 25                     jr      nz,1a5fh        ; Not a 'shell', so go on ahead
1A3A  36 20                     ld      (hl),' '        ; Get rid of the comma
1A3C  11 101B                   ld      de,1b10h        ; Display 'shell wait' message
1A3F  0E 09                     ld      c,9             ; (note: message is trashed by other code and
1A41  CD 0500                   call    0005            ; ..cannot be called from elsewhere)
1A44  CD 491A                   call    sak             ; Wait for key to be pressed
1A47  18 16                     jr      1a5fh           ; Proceed normally
1A49                    sak:
1A49  1E FF                     ld      e,0ffh          ; Poll console input status
1A4B  0E 06                     ld      c,6
1A4D  CD 0500                   call    0005
1A50  B7                        or      a
1A51  28 F6                     jr      z,sak           ; ..until a key is pressed
1A53  1E 0D                     ld      e,0dh           ; Echo carriage return
1A55  0E 06                     ld      c,6
1A57  C3 0500                   jp      0005
1A5A                    end5pat:
       0500             free    defl    1a5fh - end5pat
       0000                     if      $ gt 1a5fh
                                endif
                               end
```

It is possible to use MLOAD to install just the patches for WS.COM. Simply delete the parts of WSPAT.Z80 that refer to patches in WS.OVR and assemble the remaining code to a HEX file.

Enjoy playing with (and using) this different (improved) version of WordStar 4, and let me know what you think and what further suggestions you have. ■

Registered Trademarks

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

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

(Continued from page 3)

license or royalty fees, and also protect your source code.

Predictions for 1990

What changes will we see by the end of 1990? Will IBM still reign supreme? Will the standard hard drive be 200 Megs? Will we even be using hard drives, or will they have been replaced by erasable optical drives?

Submit your predictions. Send a few words or a paragraph to be compiled with others, or send an article. Be fearless and sign your name, or send it anonymously, but send it by December 31, to be included in issue #37. Perhaps we can even get Hal Hardenbergh to share a few words of wisdom.

We need lots of input, so tell your friends, announce it at meetings, post a message on BBSs—tell them to mail their predictions to TCJ. Most importantly, send **YOUR** predictions.

Modula-2

I have wanted to establish a public domain library of functions which can be linked in to the user's program. C looked very attractive because of its library and linking features. The problem is that it is difficult to control scope and parameter passing between functions written at different times or by different people. Versions 4 and 5 of Turbo Pascal® support separate compilation and linking of units, but they make no improvement in the control of scope and parameter passing.

Either C or Turbo Pascal can work very well for a single programmer who is very diligent about controlling the scope and parameter passing between code sections which are to be linked. They do not work well where code sections created at different times or by different people are used.

Modula-2 was designed to provide the scope and parameter controls which are missing in the other languages. Its syntax is very similar to Pascal, but there are a lot of additions which minimize Pascal's weaknesses.

It appears that Modula-2 is the most suitable language currently available commercially at a very reasonable price, which can be used to establish a library of portable code sections (modules). For example, the module `CompDir`, which starts on page 4, runs on MS-DOS, CP/M, or Atari machines when it is compiled using the appropriate Modula-2 compiler.

The article by Dave Moore in this issue, is the beginning of a regular section covering Modula-2. The main coordinator for the Modula-2 section is Barry Workman (Workman & Associates 818-791-7979). The CompDir module is included in his Editor/Toolkit, and he offered to include the CompDir module with his compiler if the reader mentions this article when ordering.

Your input is needed. Send your articles and disk contributions, or tell Barry or I what you would like to see. We'll start a *Requested Module Register* if you let us know what you need.

Expert Systems

The fields of expert systems and artificial intelligence have attracted a lot of attention, but most of the work has been theoretical with few real world applications. Too many of the books give kindergarten level examples such as, "Jane has red hair, June has blond hair, Tom likes someone who has red hair. Who does Tom like?").

There is a need in business and industry for expert systems, but the programming tools have not been available. Prolog and Lisp are the commonly accepted AI and expert systems languages, but the learning curve is very steep for someone who wants to create an expert system for an urgent real world application.

Our interest is in creating expert systems for applications such as diagnosing post-polio syndrome, and expert system shells for databases. I want to create expert systems, without spending months learning Lisp or Prolog, and I have been searching for a suitable tool.

We are now evaluating 1st Class Fusion (1st Class Expert Systems, 1-800-872-8812). Fusion can create expert systems based on either rules or examples, it can read, write, or append dBASE III files, examples can be imported from an external database, and it can generate program source code for Pascal , C, or production rules. A run time program is included so that the finished advisors can be distributed royalty free.

Our initial evaluation indicates that Fusion is well suited for our applications. Tom Hilton and I are working on a more extensive evaluation, and an in-depth evaluation will be presented in a future issue.

The way people interface with computers (more properly, the way computers interface with people) will have to change as more people use computers as appliances. New software will have to use improved interfaces, and existing programs will have to be revised. Methods of accessing knowledge bases and accessing expert's knowledge must also be improved.

Opportunities in the fields of expert systems, knowledge bases, and expert system shells are wide open. We are very interested in hearing from anyone working in these fields.

Gremlins Strike

We typeset code listings directly from the author's file in order to avoid errors, but sometimes the gremlins still manage to screw things up. After issue #33 was shipped, we discovered that we were getting errors in the serial link between the computer and the typesetter (there are no provisions for error checking in the typesetter interface).

There were several transmission errors in Sage's listings on pages 36 and 37 of issue #33. The most serious was in Listing 3, where `exitsub` was shown as `03vh`. That was a transmission error. Since that is obviously not a hex number, people used the value of `03b2h` in listing 2. Unfortunately, that was an error on Sage's disk. The correct value is `03beh`. ■

Back Issues Available:

Issue Number 18:
- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1
- The Computer Corner

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

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

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

Issue Number 22:
- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column
- The Computer Corner

Issue Number 23:
- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:
- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:
- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro Little Board
- Building a SCSI Adapter
- New-DOS: CCP Internal Commands
- Ampro '186: Networking with SuperDUO
- ZSIG Column

Issue Number 26:
- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:
- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats
- The Computer Corner

Issue Number 28:
- Starting Your Own BBS: What it takes to run a BBS.
- Build an A/D Converter for the Ampro L.B.: A low cost one chip A/D converter.
- The Hitachi HD64180: Part 2, Setting the wait states & RAM refresh, using the PRT, and DMA.
- Using SCSI for Real Time Control: Separating the memory & I/O buses.
- An Open Letter to STD-Bus Manufacturers: Getting an industrial control job done.
- Programming Style: User interfacing and interaction.
- Patching Turbo Pascal: Using disassembled Z80 source code to modify TP.
- Choosing a Language for Machine Control: The advantages of a compiled RPN Forth like language.

Issue Number 29:
- Better Software Filter Design: Writing pipable user friendly programs.
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a nes OS and the 68000?
- Detecting the 8087 Math Chip: Temperature sensitive software.
- Floppy Disk Track Structure: A look at disk control information & data capacity.
- The ZCPR3 Corner: Announcing ZCPR33 plus Z-COM Customization.
- The Computer Corner.

Issue Number 30:
- Double Density Floppy Controller: An algorithm for an improved CP/M BIOS.
- ZCPR3 IOP for the Ampro L.B.: Implementing ZCPR3 IOP support featuring NuKey, a keyboard re-definition IOP.
- 32000 Hacker's Language: How a working programmer is designing his own language.
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part two.
- Non-Preemptive Multitasking: How multitasking works, and why you might choose non-preemptive instead of preemenptive multitasking.
- Software Timers for the 68000: Writing and using software timers for process control.
- Lilliput Z-Node: A remote access system for TCJ subscribers.
- The ZCPR3 Corner
- The CP/M Corner
- The Computer Corner

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

Issue Number 32:
- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.

(Continued)

• 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 make these chips 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.

TCJ ORDER FORM

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

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

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____ State_____ ZIP _____

The Computer Journal

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

Special Close Out Sale on these back issues only

Issue Number 1:
- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an "Epram"

Issue Number 2:
- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:
- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for the Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:
- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

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

Issue Number 8:
- Build VIC-20 EPROM Programmer
- Multi-User: CP/Net
- Build High Resolution S-100 Graphics Board: Part 3
- System Integration, Part 3: CP/M 3.0
- Linear Optimization with Micros

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

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

Issues—1, 2, 3, 4, 6, 8, 14, 15, and 16
3 or more, $1.50 each postpaid in the U.S.
Outside of the U.S., 3 or more, $2.50 each postpaid surface.
List your second choice if possible.
Subject to the supply on hand.
Other back issues are available at the regular price.

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

Use TCJ Order Form

THE COMPUTER CORNER

by Bill Kibler

Well, this month catches me as busy as ever, and unfortunately not with computers. I just completed the process of moving, and getting started on rebuilding the computer room. I do have some topics to comment on, like the SOG.

The SOG is a once a year get together in Oregon, and it is usually a time for the Computer Journal staff to meet face to face. This year was a bit different as the meeting didn't have the use of the colleges dorms, and I feel that the lack of dorms cut down on the number of people as well as all around activities.

The attitude of most people there was rather low key, and I found few of the talks worth going to. A few of us talked this over and had mixed feelings about what was going on. My personal feelings are based on my personal situation as well as my impressions of the industry as a whole.

A Sogger's Profile

To understand the people that go to the SOG, it might help to create a profile of what a typical SOGee is. Most SOGGERs have been working with computers longer than MSDOS has been around, in fact most probably remember IBM's two previous failures to enter the micro market. I dare say that most have anywhere from 3 to 10 computers, either at home or at least at their disposal. They probably use and understand two or three operating systems. Dealing with hardware is a fact of life for them, as well as assembly language programming.

For most of us the SOG has previewed many new ideas and products. Some of those ideas have had impact on the industry, while others haven't been seen again. With that profile in mind and the idea that new products are a feature of the SOG, it is not too hard to understand why many of us were rather low key this year. I didn't see anything I would consider new or on the leading edge. Several of the innovators were absent this year, adding to the lack of spark.

Spark and controversy were definitely missing. At times I personally had a depressed outlook. I kept feeling that most SOGGERs had given up on computers. That could be a product of the PC syndrome, or just burn out. I know mine is from watching the industry shoot itself in the foot. Whatever the cause, should next year be like this year, it will be my last trip to Bend.

The Computer Media

I recently received my August issue of MINI-MICRO SYSTEMS which was devoted to UNIX and MSDOS/OS2. I stopped reading after the third article. The bias expressed in the articles was so clear to those of us who have been around for a while. The bias I talk about is the PC syndrome. Their computer revolution started with the PC and not CP/M or Apples. I found their talk about MSDOS being the only micro operating system a bit hard to take, even when they have a smaller article talking about DRI's MSDOS replacement operating system. Nowhere in any of the articles was there anything about DRI's DOS386 which is available now, and for my feelings a better system than OS/2 will ever be.

Over the years I have been saying that the only reason the PC has done as well as it did was due to media hype. I can still remember when BYTE introduced the machine with such glowing reviews. Of course later they got in trouble when people found out they were paid indirectly to say that. I still remember a conversation with a real estate agent in northern California who was waiting for IBM to do it before he would buy a computer. This person didn't know anything about computers, how they worked, or even what his real needs would be in using one. His only criteria was the name, nothing else mattered.

RTX and Other Products

I hope that one big sale does go through, that being HARRIS's buying of GE's silicon factories. As I write this GE is selling off their acquired RCA and GE's semiconductor factories. Harris is in the process of trying to get them, mostly for the government business. My interest is their RTX 2000, or a NOVIX based CPU PLUS. My NOVIX was a great turn on for most people at the SOG, but I didn't find a buyer. I am trying to sell my Plexiglas machine because I want to buy the RTX system and experiment around with it.

Harris has released more information on the RTX and in fact they have a small dog and pony show making it to many of the Forth local chapter meetings. I now have a copy of their programmer's guide, and what it can do now over the NOVIX is really impressive. The addition of a memory management unit, the two timers, and their ASIC bus make the system really smoke. I still feel this is probably one of the faster chips produced today. If you consider the time to develop a program and then the speed of the device, the product is hard to beat.

If your company or operation is using either a NOVIX or a RTX2000 we here at the Computer Journal would sure like to hear about them. My last few months have been keeping me busy and away from computers, so I need to rely on our readers for a change. I have lots of way the NOVIX can be used, but what I need most is applications to place them in.

A Good Book

About the only good item I picked up at the SOG was a book by Bruce Eckel. Bruce is a writer for MicroC and has written several articles on using computers to turn things on and off. Most of his articles are tutorial in nature and have both hardware and software features. The first day of the SOG his shipment of books (he is self publishing) came just in time to start selling them. The title is *COMPUTER INTERFACING WITH PASCAL & C, Hardware + Software Projects on your PC*. For $49.90 you can get both his book and the accompanying disk of software source code.

I have been reading the book and find it very good for beginners. The book is mostly his older articles all complied in one place. For advanced hackers the book would be easy reading and of limited use. I find the code samples of some use for

(Continued on page 17)