The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 31

$3.00

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

ISSN # 0748-9331

The COMPUTER JOURNAL

Features

Issue Number 31

Columns

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

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Production Assistant
Judie Overbeek

Circulation
Donna Carlson

Contributing Editors
Joe Bartel
Bob Blum
Bill Kibler
Rick Lehrbaum
Frederick B. Maxwell
Jay Sage
Kenneth A. Taschner

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

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

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

MOVING?

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

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

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

Registered Trademarks

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

Apple II, II +, IIc, IIe, Macintosch, DOS 3.3, ProDOS; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. MBASIC; Microsoft. Wordstar; MicroPro International Corp. IBM-PC, XT, and AT; IBM Corporation. Z-80, Zilog. MT-BASIC, Softaid, Inc. Turbo Pascal, Borland International.

Where these terms (and others) are used in The Computer Journal they are acknowledged to be the property of the respective companies even if not specifically mentioned in each occurence.

Editor's Page

Is the PC-XT/AT the Next Orphan?

IBM has traditionally introduced new mainframe models with revisions intended to force their customers to replace equipment with the newest models — and then phased out support for the older systems. Now, they're trying the same approach with their PC.

IBM has discontinued production of both the PC-XT and the PC-AT, in favor of the PS/2, with the intention that all current software and all new software will be written for the PS/2, and will not run on the older systems. If they can accomplish this, and at the same time prevent the cloning of the PS/2, they would stamp out the clones which are taking their market share.

But, there are millions of PC-XT/ATs out there, and they provide a very attractive market for third party support. The PC market is a lot different than the CP/M market, because the CP/M market was badly fragmented by the many manufacturers who all wanted to do it their way. You couldn't just buy a CP/M program and expect it to run in your CP/M system because of the various disk sizes, disk formats, I/O provisions, BIOS implementations, and terminal configurations. You had to patch and install the software, and sometimes the program made assumptions which prevented it from running on your system. The PC-Clone market is different, because the machines wouldn't sell unless they were highly PC compatible, so any program will run in any system — except for some graphics or extended memory requirements which can be met by buying a relatively inexpensive card.

The XTs and ATs are poorly designed and outdated. It is obvious that they will have to be replaced in applications requiring more speed and power. The question is, what will replace them? We have skipped the XT generation, and will probably skip the AT generation, while we keep hacking on our CP/M systems, our AMPRO '186 Little Board, and the Hawthorne 68000 Tiny Giant. When we really need something for CAD or large databases, we'll see what else has developed. We'll locate or write the software, and then buy the hardware that fits. Right now the Compaq 386/20 looks very interesting, and appears better that anything that IBM has to offer. The cloners just might beat IBM at their own game!

Viewpoints on System Design

I am fully aware that I am nonconventional, and that no one will ever produce an affordable system which will satisfy me, but that does not prevent me from forming an opinion of what I want. As Charles McCabe from the *San Francisco Chronicle* said, "Any clod can have the facts, but having opinions is an art."

> **"Any clod can have the facts, but having opinions is an art."**
>
> Charles McCabe
> San Francisco Chronicle

I'm preparing a series on "What a hardware/operating system should provide," and your input in the form of letters, notes, comments, articles, etc. are welcome.

Perhaps we'll start a section called 'Viewpoints" where you can sound off about hardware, operating system, or software topics.

Some of my requirements are: 1) An open Bus, 2) Modifiable Operating System with source code, 3) Extensive, easily expanded I/O (at least four serial ports and two bidirectional parallel ports) with the ability to redirect the I/O to any port from within a program, 4) The ability to add slave boards which can be anything from complete Single Board Computers to specialized numeric control boards, 5) The ability to use different CPU families at the same time, 6) The ability to add almost any quantity or type of peripheral.

If I could retire and have lots of time, I'd consider hacking a system based on an S-100 mother board using a redefined Bus. What you'd like to see in a system is different than what I want, so send your input and make yourself heard.

More on User Interfacing

In the past several issues, I have been talking about the problems in designing and implementing the user interface. So far, I have come up with more questions than answers, but at least I am getting a feel for what I want to do.

A large part of the problem is the lack of published realistic, working-level, information. The lack of information is easy to understand because: 1) The sub-

C CODE FOR THE PC

source code, of course

C Source Code

Bluestreak Plus Communications (two ports, programmer's interface, terminal emulation)	$400
CQL Query System (SQL retrievals plus windows)	$325
GraphiC 4.1 (high-resolution, DISSPLA-style scientific plots in color & hardcopy)	$325
Barcode Generator (specify Code 39 (alphanumeric), Interleaved 2 of 5 (numeric), or UPC)	$300
Greenleaf Data Windows (windows, menus, data entry, interactive form design)	$295
Vitamin C (MacWindows)	$200
resident C (TSRify C programs, DOS shared libraries)	$165
Essential C Utility Library (400 useful C functions)	$160
Essential Communications Library (C functions for RS-232-based communication systems)	$160
Greenleaf Communications Library (interrupt mode, modem control, XON-XOFF)	$150
Greenleaf Functions (296 useful C functions, all DOS services)	$150
OS/88 (U**x-like O/S, many tools, cross-development from MS-DOS)	$150
Turbo G Graphics Library (all popular adapters, hidden line removal)	$135
CBTree (B+tree ISAM driver, multiple variable-length keys)	$115
MultiDOS Plus (DOS-based multitasking, intertask messaging, semaphores)	$115
PC/IP (CMU/MIT TCP/IP implementation for PCs)	$100
B-Tree Library & ISAM Driver (file system utilities by Softfocus)	$100
The Profiler (program execution profile tool)	$100
Entelekon C Function Library (screen, graphics, keyboard, string, printer, etc.)	$100
Entelekon Power Windows (menus, overlays, messages, alarms, file handling, etc.)	$100
Wendin O/S Construction Kit or PCNX, PCVMS O/S Shells	$95
QC88 C Compiler (ASM output, small model, no longs, floats or bit fields, 80+ function library)	$90
JATE Async Terminal Emulator (includes file transfer and menu subsystem)	$80
MultiDOS Plus (DOS-based multitasking, intertask messaging, semaphores)	$80
ME (programmer's editor with C-like macro language by Magma Software)	$75
WKS Library (C program interface to Lotus 1-2-3 program & files)	$65
Quincy (interactive C interpreter)	$60
EZ_ASM (assembly language macros bridging C and MASM)	$60
PTree (parse tree management)	$60
HELP (pop-up help system builder)	$50
Multi-User BBS (chat, mail, menus, sysop displays; uses Galacticomm modem card)	$50
Heap Expander (dynamic memory manager for expanded memory)	$50
Make (macros, all languages, built-in rules)	$50
Vector-to-Raster Conversion (stroke letters & Tektronix 4010 codes to bitmaps)	$50
Coder's Prolog (inference engine for use with C programs)	$45
C-Help (pop-up help for C programmers ... add your own notes)	$40
Biggerstaff's System Tools (multi-tasking window manager kit)	$40
CLIPS (rule-based expert system generator, Version 4.0)	$35
TELE Kernel (Ken Berry's multi-tasking kernel)	$30
TELE Windows (Ken Berry's window package)	$30
Clisp (Lisp interpreter with extensive internals documentation)	$30
Translate Rules to C (YACC-like function generator for rule-based systems)	$30
6-Pack of Editors (six public domain editors for use, study & hacking)	$30
ICON (string and list processing language, Version 6 and update)	$25
LEX (lexical analyser generator)	$25
Bison & PREP (YACC workalike parser generator & attribute grammar preprocessor)	$25
AutoTrace (program tracer and memory trasher catcher)	$25
C Compiler Torture Test (checks a C compiler against K & R)	$20
Benchmark Package (C compiler, PC hardware, and Unix system)	$20
TN3270 (remote login to IBM VM/CMS as a 3270 terminal on a 3274 controller)	$20
A68 (68000 cross-assembler)	$20
List-Pac (C functions for lists, stacks, and queues)	$20
XLT Macro Processor (general purpose text translator)	$20
C Tools (exception macros, wc, pp, roff, grep, printf, hash, declare, banner, Pascal-to-C)	$15

Data

WordCruncher (text retrieval & document analysis program)	$275
DNA Sequences (GenBank 48.0 of 10,913 sequences with fast similarity search program)	$150
Protein Sequences (5,415 sequences, 1,302,966 residuals, with similarity search program)	$60
Webster's Second Dictionary (234,932 words)	$60
U. S. Cities (names & longitude/latitude of 32,000 U.S. cities and 6,000 state boundary points)	$35
The World Digitized (100,000 longitude/latitude of world country boundaries)	$30
KST Fonts (13,200 characters in 139 mixed fonts: specify TEX or bitmap format)	$30
USNO Floppy Almanac (high-precision moon, sun, planet & star positions)	$20
NBS Hershey Fonts (1,377 stroke characters in 14 fonts)	$15
U. S. Map (15,701 points of state boundaries)	$15

The Austin Code Works
11100 Leafwood Lane
Austin, Texas 78750-3409 USA
acw!info@uunet.uu.net

Voice: (512) 258-0785
BBS: (512) 258-8831
FidoNet: 1:382/12

Free surface shipping on prepaid orders

MasterCard/VISA

ject is very broad with widely differing requirements for different applications, so that there is not one simple answer. 2) The people who have the experience and knowledge to provide the answers are too busy to take the time to write.

I feel that designing user interfaces is an art rather than a science, and that we have to experience a wide range of existing products running on many different systems in order to develop a intuitive feeling for what will work. We also need to talk to other developers and share our thoughts on what works, what doesn't work, and how to implement our ideas. We should first concentrate on generic ideas which are not system specific, because we'll all be working on many different systems in the future. After we know what we want, we'll talk about how it can (or perhaps how it can't) be accomplished on the various systems.

An example of good and bad user interface which I ran into this week is how a program reacts to the printer being off line when the program tries to access it. Working with ZCRP3 on an AMPRO Z80 Little Board, WordStar® CP/M Edition Release 4.0 allows me to enter a Control-U to return to WordStar. An older version of CalcStar waits for a while, and then returns to the program with a message about checking to be sure the printer is ready. Both of these are fine, but I had another program which locked up and would not return even if the printer was later placed on line — you had to reboot the system which meant that you lost what ever you were doing. I would normally think of implementing this through a BIOS modification, but obviously it can be done through the program, and I'll have to figure out how to do this in various languages (HINT: Tell us how you do it with your systems).

One of the things which has been delaying the user interface project is that I have discovered that I am not adept at parsing the user input. I especially have trouble providing for the variations and errors in the input. Every time I get started on this project, I find that I have to start somewhere else first. Perhaps this is an example of top-down design, where I first decide what I want to do, then decide what routines I'll need, then decide what routines those routines will need, then decide what routines... I'm making great progress backwards!

Right now, I'm off on a side track investigating the use of YACC (Yet Another Compiler Compiler) and LEX (LEXical Analyzer) to provide the parsing routines for C programs [LEX plus BISON (a YACC workalike) are available with C source code for the IBM PC from Scott Guthery, The Austin Code Works, 11100 Leafwood Lane, Austin, Texas 78750-3409, phone (512) 258-0785]. I'd like to know if it is practical to use LEX and YACC to generate portable parser routines which can be compiled under CP/M, PC DOS, or 68000 systems.

Programming Tools

Programmers and system implementors need a highly coordinated toolkit with much more than just the normal editors, compilers, and assemblers. ZCPR33 provides an amazing amount of power from a Z80, but there are times when I need more power than is available from an 8-bit system. I have never been comfortable with PC/MS DOS, because it is awkward with many utilities and TSR (Terminate and Stay Resident) programs which don't work well together — you can do almost anything, but you have to do it one step at a time with different programs and it is difficult to pipe data from one program to another.

Two of our boys made it home for Christmas this year, and we spent a week enjoying the winter splendors in nearby Glacier National Park and the surrounding mountains, plus hot and heavy debates (arguments??) about computers. Dave is very aware of my dislike for PC-DOS, so he demonstrated The MKS Toolkit (Mortice Kern Systems Inc., 35 King Street North, Waterloo, Ontario, Canada, N2J 2W9 (519) 884-2251). Their ad states "Spans both worlds UNIX® — DOS," and while I am not currently interested in a UNIX machine, and I don't like PC-DOS, I do like the combination of the MKS Toolkit running under PC-DOS. The toolkit includes the UNIX VI/EX editor, the KORN Shell, AWK, and over 110 UNIX commands such as cpio, find, sum, tr, tee, gres, fmt, login, ls, lc, etc. Dave demonstrated some of the utilities and how to write scripts operating under the shell, and then I took the computer away from him and stayed up late enjoying the toolkit. Dave got even for my taking away the computer by taking the toolkit back with him. It's only $139, and I'll have to get a copy of my own. Incidentally, it works fine on my AMPRO '186 Little Board under PC-DOS Version 3.10 using a Zenith Z19 ASCII terminal — none of the distracting graphics which I hate. If you use a PC for anything other than canned commercial programs, you owe it to yourself to take a look at the MKS Toolkit.

The C Users' Group Expands

The C Users' Group has acquired *The C Journal*, and combined it with *The C Users' Group Newsletter*, to form *The C Users Journal*. Robert and Donna Stucy Ward are to be congratulated on the first issue of their new publication. It is now even more important for anyone interested in programming in C to join The C Users' Group to get their journal and access to the extensive CUG disk library. You can contact them at The C Users' Group, P.O. Box 97, McPherson, KS 67460, phone (316) 241-1065.

Their articles on LEX and YACC are very timely, because that's what I am currently working on. ■

Feedback Reader's

HD64180 MMU Feedback

John Schneider's article in TCJ #27 on the HD64180 is very informative and useful, but I have just discovered an error in it that I would like to share with the readers. It concerns the section on the Memory Management Unit (MMU), in particular, the Common Base Register (CBR).

Schneider states that if "You want to switch a block of memory starting at physical address 50000h into the Common Area 1 segment (at the same time switching out whatever was there)...the CBR register would be loaded with 50h. It's that simple."

No it isn't. That statement is only true if a CBAR value of 0 is used. (That is the Common Area 1 takes up the entire logical area beginning at logical 0000.) For example, if the mapping shown further down on P.26 is in effect where the common area 1 occupies the upper 32K of logical memory and the bank area the lower 32K, then a CBR value of 50h will load physical memory from 58000h to 5FFFFh into the logical space from 8000h to FFFFh. The rule is that the CBR value is ADDED to the upper 4 bits of the logical address and this 7-bit value forms the upper 7 bits of the 19-bit phyiscal address. The other twelve bits are simply the lower 12 bits of the logical address.

The same rule applies to the BBR, although in most typical applications the Base area starts at logical 0, so the method Schneider gives will work.

So then, to complete the example, if we want a Common Area beginning at Logical address 8000h to contain physical addresses starting at 50000h, we must put the value 48h into the CBR. 48 h is added to 8h (the upper 4 bits of the logical address) to yield 50h, which will load the physical addresses from 50000h into logical memory at 8000h.

Steve Cohen

Editor's Note: Steve, thanks for the feedback. One of the primary purposes of TCJ is to provide a forum for our readers to exchange information.

Consulting In the Real World

I am looking forward to the Lilliput Z-NODE. Some other BBS's I use are the Morrow Owners Review BBS at 1-415-654-3798, BAMDUA (Bay Area Micro Decision Users Association, another Morrow Group) at 1-415-948-2513, PRACSA at 1-415-948-2513 (this is run by Irv Hoff, who wrote IMP), Turbo Source Search at 1-617-545-9131 (this is a BBS that specializes in Source Code), and an MS-DOS oriented BBS, the Software Society, at 1-201-729-7410.

I appreciate being abreast of the trends in hardware and software. I do mostly financial applications programming. I have freelanced on and off for about four years. My primary languages that I use are basic, and Turbo Pascal.

I appreciated Bill Kibler's column of PC's, sales of PC's lack of training, and the importance of users to outline their needs. I have been directly in that type of position. It doesn't just happen with PCs. I had a pension planning firm try to use a NorthStar computer running three users under TSS/C, (NorthStar's version of M/PM) to do all of their pension accounting for their clients. Bill's suggestion that the client write-up what they need sounds really good, but it doesn't work that way. Most clients don't know enough about computer systems to write-up clearly what they want, if they did they wouldn't be going to a consultant in the first place. Clients can get hold of demo packages relatively cheaply. Often though, the clients don't even understand the demo packages, because they have never worked with a computer system before, or their experience is limited to a fixed pattern. I had a bookkeeper ask me to explain an accounting package, which I did to the best of my ability, then it was the bookkeeper's turn to come up with a chart of accounts. This was completely beyond the bookkeeper's ability. This is not a computer problem, this is a training problem or lack of skills, that gets shuffled off by saying the computer can't do it, or that the programmer can't explain it simply. I could have done the chart of accounts, but I wasn't going to, because I didn't want to run over the accountants. Also, clients are not always as innocent as Bill states they are. Clients will try to use two consutants at the same time and juggle one against the other. Clients also try to have the consultant do the biggest jobs first, rather than starting with small jobs and working up. They also want consultants to do everything at once.

I would like to see more MS-DOS articles, because most of my work is in PCs. I have a Morrow at home, but it doesn't pay the bills.

R.U.

Editor's Note: Thank you for sharing your experiences. How about more of you sharing your experiences?

There are so many publications covering the PCs and MS-DOS, that I felt that there wasn't much we could add. The fact is that we have to work with the PCs to pay the bills, and I am not entirely against including technical PC topics which are not duplicated elsewhere. Specific suggestions on PC topics will be welcome.

I understand that the Morrow Owners Review is defunct, and that their BBS may be closed or possibly transfered to another location. Does anyone have any information on this?

Wants "Case History"

My wife and I have three computers in use. A Macintosh SE, an Apple IIe, and an SB-180FX which runs Echelon's Z-System software.

I enjoy the hardware oriented articles the best, and material on using assembly language as well.

What would I like to see in TCJ? How about some "case History" type articles on how someone has incorporated an Ampro Little Board, SB180 or one of the Hawthorne 68K boards into a "real" project. Emphasis on the "real" and not just what the marketing types claim can be done. Talk about the software as well. It seems like one of these boards would be a

(Continued on page 41)

Using SCSI for Generalized I/O

SCSI Can be Used for More Than Just Hard Drives

by Rick Lehrbaum, Vice President Engineering, Ampro

SCSI Catches On!

Over the past two years, the Small Computer System Interface ("SCSI") has begun to be included as a standard feature in the microcomputer pr
Apple) and board manufacturers (such as AMPRO). This is a result of three factors:

(1) SCSI has finally been approved by the American National Standards Institute (ANSI X3.131).

(2) Single chip SCSI interface IC's such as the NCR 5380 have become common and inexpensive. (The 5380 already has at least five alternate sources.)

(3) Hard disk drives such as the Seagate 225N and tape drives such as the Teac MT2ST are now available with "embedded" SCSI controllers.

Thanks to the ease of integration and very low cost of including a SCSI interface (due to devices like the 5380), designers of microcomputer products (systems and boards) now routinely include a SCSI bus controller.

Another Way to use a SCSI Port

In a lot of data acquisition and control or embedded microcomputer applications, the SCSI port may go unused. If your system's SCSI bus is not required for "normal" SCSI device connection, you may be able to use the SCSI interface port as a generalized I/O interface instead.

The ability of a SCSI interface to be used for other types of I/O depends entirely on the hardware that is being used to generate the SCSI bus signals. Some SCSI interface IC's are quite "intelligent," while others are relatively "dumb." In general, because the dumb SCSI IC's require lower level control by the system CPU, they provide more direct CPU control over each of the SCSI interface signals than do the smarter IC's. Therefore, the dumber the SCSI IC, the more likely it is to be useful as a programmable I/O port. On the other hand, some of the smarter SCSI IC's are too specialized to allow this flexibility.

This article discusses the use of a 5380 SCSI controller IC as a generalized I/O interface. The 5380 offers nearly total control over the 17 signals which comprise the SCSI bus. Although the 5380 was not designed to serve as a general purpose I/O port, it has several important features which make it well suited for this purpose:

- Open collector output buffers, with 48 mA current sink capability.
- Schmidt-trigger conditioning on input buffers.
- Simple CPU bus interface with DMA logic.
- Seventeen software controlled I/O signals.
- Handshake and interrupt logic (usable in some applications).

Inside the 5380

The 5380 has 17 bidirectional I/O lines, which may be used as inputs or outputs under software control. It also offers several more advanced features including interrupts, request/acknowledge handshaking, and DMA support. These advanced features are intended specifically for SCSI, so they are not very flexible; however you may find one or more of them useful in a particular application.

To fully understand the 5380 SCSI Protocol Controller device, you should obtain a copy of the NCR 5380 Design Manual, available for a nominal charge from NCR (see reference below). In this article, we will only focus on the simple I/O functions.

Within the 5380 are eight readable and eight writable internal ports, normally addressed as eight consecutive I/O addresses. What follows next is a brief description of the function of each of the 5380's internal registers. The I/O addresses indicated are the normal offsets from the 5380's base address in your system. Note that all of the SCSI bus signals (at the 5380 IC's pins) are "active low," so the actual bus voltage levels are opposite to the contents of the corresponding bits in the 5380 registers.

SCSI Data Register (00, read/write): Writing to this register in the 5380 sets the state of the SCSI bus data lines (DB0 through DB7), providing that the "Assert Data Bus" bit of the Initiator Command Register is set. If you write to this register when the Assert Data Bus bit is not set, the register will hold your data but not assert it on the SCSI bus until the Assert Data Bus bit (in the Initiator Command Register) is set at a later time. The SCSI Data Register's data bits are assigned as follows:

```
 Bit 7     6        5        4        3        2        1      Bit 0
+--------+--------+--------+--------+--------+--------+--------+--------+
:  DB7   :  DB6   :  DB5   :  DB4   :  DB3   :  DB2   :  DB1   :  DB0   :
+--------+--------+--------+--------+--------+--------+--------+--------+
                          SCSI Data Register
```

When you read this I/O port, the value obtained represents the current state of the SCSI bus data lines, DB0 through DB7, except that the actual voltages on the bus lines are inverted relative to the contents of this register.

Initiator Command Register (01, read/write): This register is primarily used to control the 5380's SCSI bus interface when the chip is in the Initiator role. Most functions are also available in the Target role. Two of the bits of this register have different uses when the register is read or written, so two charts are given. These are as follows:

```
 Bit 7     6        5        4        3        2        1      Bit 0
+--------+--------+--------+--------+--------+--------+--------+--------+
:Assert  : Arb in: Lost    :Assert  :Assert  :Assert  :Assert  :Assert  :
: RST    : Prog   :  Arb   : ACK    : BSY    :  SEL   : ATN    : Data   :
+--------+--------+--------+--------+--------+--------+--------+--------+
               Initiator Command Register -- Read Usage

 Bit 7     6        5        4        3        2        1      Bit 0
+--------+--------+--------+--------+--------+--------+--------+--------+
:Assert  : Test   : Diff   :Assert  :Assert  :Assert  :Assert  :Assert  :
: RST    : Mode   :  En    : ACK    : BSY    :  SEL   : ATN    : Data   :
+--------+--------+--------+--------+--------+--------+--------+--------+
               Initiator Command Register -- Write Usage
```

As you have probably guessed, the Initiator Command Register allows you to control the state of the RST, ACK, BSY, SEL, and ATN bus signals, and also to control whether the 5380 places its data on the SCSI bus or not. Notice that bits 6 and 5 differ according to whether you are reading or writing this register. (Refer to the 5380 Design Manual for details on the use of these bits.)

Here are three restrictions in using these bits to control the SCSI bus:f

(1) The 5380 must be in Initiator Mode (Mode Register, bit 6) to be able to set the SCSI control bits ACK and ATN active on the SCSI bus.

(2) If the 5380 is in Initiator Mode (Mode Register, bit 6), then the data bus will not be asserted by the Assert Data Bus bit (Bit 0) unless the SCSI bus I/O signal is false (output from Initiator) and the SCSI bus control signals C/D, I/O, and MSG all match the contents of the Assert bits in the Target Command Register.

(3) When the Assert RST bit is set, the resulting RST signal on the SCSI bus clears all of the 5380's internal registers! (Not a very useful general purpose signal, is it?)

Mode Register (02, read/write): This register contains many control signals governing operation of the 5380. It allows you to place the chip in either Initiator or Target mode, and provides control over DMA and arbitration functions, parity, etc.

Bit 7	6	5	4	3	2	1	Bit 0
Block Mode DMA	Target Mode	Enable Parity Check	Enable Parity Int	Enable EOP Int	Monitor BSY	DMA Mode	Arbi-trate

Mode Register

This article will not cover the use of the bits regarding DMA, parity, arbitration, and interrupts, as these are not required for basic operation of the SCSI interface. Bit 6 is the most interesting bit of this register, because it determines whether the 5380 is in Target Mode or Initiator Mode.

Target Command Register (03, read/write): This register provides control over the bus phase control bits: REQ, MSG, C/D, and I/O, as follows:

Bit 7	6	5	4	3	2	1	Bit 0
				Assert: REQ	Assert: MSG	Assert: C/D	Assert: I/O

Target Command Register

These bits can only be asserted by the 5380 if the "Target Mode" bit in the Mode Register is set. In Initiator mode, these bits have a different purpose. In Initiator Mode, the states of the Assert MSG, Assert C/D, and Assert I/O bits must match the actual state of the bus (which can be read in the Current SCSI Bus Status Register), for data to be placed on the SCSI bus even if the Assert Data Bus bit of the Initiator Command Register is set. Also, in Initiator Mode, if the Assert MSG, C/D, and I/O bits do match the bus state, then the "Phase Match" bit in the Bus and Status Register will be set.

Select Enable Register (04, write): This write-only register is used as a mask in Target Mode operation to allow the 5380's built-in selection response logic to generate an interrupt. Refer to the 5380 Design Manual for more info.

Bit 7	6	5	4	3	2	1	Bit 0
DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0

Select Enable Register

Current SCSI Bus Status Register (04, read): This read-only register allows you to read the current state of eight control signals on the SCSI bus. The bits are utilized as follows:

Bit 7	6	5	4	3	2	1	Bit 0
RST	BSY	REQ	MSG	C/D	I/O	SEL	DBP

Current SCSI Bus Status Register

DMA Control Ports (05-07, write): These are not registers but rather are used as control signals by the 5380's internal DMA logic. A write operation to one of these three I/O addresses is used as a trigger to begin the corresponding DMA mode (Send, Target Receive, or Initiator Receive). Refer to the 5380 Design Manual for more information on the use of DMA.

Bus and Status Register (05, read): This read-only register allows you to read two SCSI bus signals—ATN and ACK—which are not included in the Current SCSI Bus Status Register. In addition, six 5380 status flags which are associated with the optional use of interrupts are read through this register. The bits of this register are utilized as follows:

Bit 7	6	5	4	3	2	1	Bit 0
End of DMA	DMA Request	Parity Error	Inter-rupt Request	Phase Match	Busy Error	ATN	ATN

Bus and Status Register

As mentioned above, the use of DMA and interrupts is not covered in this article. The "Phase Match" bit is handy, in that it shows in a single bit whether the SCSI bus phase matches the settings of the Assert bits (MSG, C/D, and I/O) in the Target Command Register. The Phase Match bit is only meaningful, however, when the 5380 is in its Initiator Mode ("Target Mode" bit = 0).

The Busy Error bit is set if the Monitor Busy bit in the Mode Register has been set and if the SCSI bus BSY signal becomes false. If this occurs, the 5380 output drivers all become disabled.

Latched Data Register (06, read): Reading this register returns the latched—not current—state of the SCSI data lines. Data is latched either during a DMA Target Receive operation when ACK (pin 14) goes active, or during a DMA Initiator Receive when REQ (pin 20) goes active. The DMA Mode bit in the Mode Register must be set before data can be latched in this register. This register may also be read under DMA control using the 5380's DMA control lines. The contents of this register are:

Bit 7	6	5	4	3	2	1	Bit 0
DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0

Latched Data Register

Reset Parity Interrupt (07, read): A read of this address is used as a trigger to clear a parity error interrupt.

Simple I/O

As indicated above, the 5380 has two operating modes—Initiator Mode and Target Mode—and in Initiator Mode several constraints govern whether or not data from the 5380 can be placed on the SCSI bus signals. If the 5380 is used in the Target mode, however, these constraints are not applicable. Consequently, the 5380's Target mode results in more flexible operation for simple programmable digital I/O.

The 5380 is placed in Target mode by writing 40h to its Mode Register. Once in Target mode, fourteen of the chip's SCSI bus I/O signals can be used as bidirectional lines with either input or output capability, and two additional lines can be used as input-only lines. Table 1 gives the breakdown.

Table 1

5380 Target Mode Usage

5380 Register	Signals	Function
SCSI Data Register	DB0-DB7	Bidirectional
Target Command Register	I/O,C/D,MSG,REQ	Bidirectional
Initiator Command Register	BSY,SEL	Bidirectional
Initiator Command Register	ACK,ATN	Input only

As indicated in Table 1, ACK and ATN are inputs only in the 5380's Target Mode of operation. All of the other SCSI signals except RST can be used as either inputs or outputs. RST is unique in that it clears all of the registers within the 5380 whenever it becomes active for any reason (including being set by the 5380 itself!). In most applications you will probably want to avoid using the RST signal entirely—but be sure it is terminated along with the other I/O interface signals.

The data lines (DB0-DB7) are only enabled as outputs when bit 0 ("Assert Data Bus") of the Initiator Command Register is a 1. However, the state of the DB0-DB7 lines can be read whether the Assert Data Bus bit is true (1) or not. Since the data lines are Open Collector, they can be switched from output to input functions simply by writing all 0's to the SCSI Data Register. (The chip's outputs are inverted, so setting a data bit to 0 turns the output driver off.)

It is also possible to utilize the 5380's internal interrupt, REQ/ACK, and DMA support logic. For example, one 5380 user has taken advantage of the chip's handshake and interrupt functions to monitor the data transmitted by a computer's Centronics printer port, using the 5380 as an interrupting 8-bit input port.

In many 5380-based SCSI systems, there are additional input signals intended for the reading of SCSI Initiator ID jumpers. For example on the AMPRO Little Board single board computers, up to eight additional input bits are available in this manner if the 5380 is not being used as a SCSI port. If available, these extra input signals can be used to augment the signals provided by the 5380, thereby adding up to eight additional input lines.

Along the same lines, don't overlook an unused parallel printer port as a source of eight more buffered outputs and one or more output and input handshake signals.

As you can see, a 5380 SCSI interface provides quite a few I/O signals. The 48 mA output drive capacity allows long wire lengths, and also can be used to drive both mechanical and solid state relays.

Using Opto-22 I/O Modules

Opto-22® manufactures several types of "Mounting Racks" into which you can plug optically isolated input and output modules. Each module functions as either a single input bit or a single output bit, and the modules are available in both AC and DC versions. Voltages of up to 240 volts DC or AC can be switched or sensed, and the modules provide 4,000 volts isolation!

Opto-22's Mounting Racks hold either 4, 8, 16, or 24 optically isolated modules, and have model numbers PB4, PB8, PB16, and PB24, respectively. Since the 5380 provides a maximum of sixteen interface signals (as shown in Table 1), a single 5380 could interface with up to 16 such I/O modules, using a PB16 Mounting Rack.

To interface a 5380 with the Mounting Rack's optically isolated input or output modules, simply connect each SCSI bus signal (from the 5380) to an appropriate pin on the Mounting Rack's edgecard connector. This can be done by constructing a custom "scramble-wired" cable, or you can use a small customizable adapter card made by Opto-22 for this purpose, the Model UCA3. The UCA3 can accept a 50-pin header edgecard connector from the SCSI side, and plugs directly into the Opto-22 Mounting Rack. The UCA3 has user-programmed connection between the input and output bus sides—that is, it provides two 50-pin headers with wire-wrap posts which you wire to suit your needs.

The Opto-22 Mounting Rack can accommodate a mixture of input and output modules on the same rack. It is even possible to have a combination of input and output modules connected to the eight SCSI data lines (DB0-DB7) at the same time. To allow some of the data lines to function as inputs while others function as outputs (at the same time), keep the Assert Data Bus bit in the 5380's Initiator Command Register active at all times, and write 0's to any bits in the SCSI Data Register that are to be used as inputs. Because the 5380's outputs are open collector (and active low), a bit which is a 0 will not drive the bus at all, leaving the corresponding data line free to be driven by an input module.

Controlling an IC

Using the signals illustrated in Table 1 creatively, you can even hook them up directly to other IC's. You can redefine any signal as any desired function. For example, some signals can function as address signals, others as control signals, still others as data signals.

Many IC's are not too fussy about timings as long as minimum setup and hold times are provided. Using a technique known as "bit banging," you can easily satisfy a device's setup and hold requirements.

Interfacing to a Typical LSI Device.

As an example, a typical LSI device (such as a UART) might be interfaced to a 5380 as shown in Table 2.

Table 2

LSI Device Interface

Device Pin	Function	SCSI Signal Used
D0-D7	Data In/out	-DB0 through -DB7
A0,A1,A2	Internal register addressing	-I/O,-C/D,-MSG
-RD,-WR	Read and write (active low)	-SEL,-BSY
-CS	Chip select (active low)	-REQ

Before going on, a word about logic levels. The SCSI bus uses active low logic levels (i.e., a "0" is the high voltage level and a "1" is the low voltage level). Assuming that the LSI device is "normal," it probably requires active high data and address inputs, but active low control signals (-RD,-WR,-CS). Since the 5380 will make everything active low, the data and address values written to the 5380's registers must be inverted prior to writing to such a device.

In this example, the following sequence might be used for writing to a register within the LSI device:

(1) Write a 40h to the Mode Register, to place the 5380 in Target Mode.

(2) Invert the LSI device register address, and then write it to the I/O, C/D, and MSG bits in the Target Command Register while also setting the REQ bit (chip select) to 1.

(3) Invert the data to be written, and then write it to the SCSI Data Register.

(4) Enable data output by writing an 01h ("assert data bus") to the Initiator Command Register.

(5) Turn on the -WR signal by writing an 05h ("REQ" with "assert data bus") to the Initiator Command Register.

(6) Remove the -WR signal by once again writing an 01h to the Initiator Command Register. This provides write data hold time.

(7) Remove the chip select and address by writing 00h to the Target Command Register.

(8) Disable data output by writing 00h to the Initiator Command Register.

You will want to modify this procedure slightly, based on the actual requirements of the particular LSI device you need to control. A similar process is used to read the device.

Synthesizing a BUS

Another interesting and potentially powerful use of a 5380 SCSI interface is in mimicking the functions of a bus. Although you can't expect to generate anything as complex as a Multibus or VME bus, there are several simple I/O-oriented buses which can be synthesized adequately using just the 5380 and a scramble-wired cable between the 5380 and the bus cards or backplane. Two bus interface examples follow.

Interfacing to the "A-Bus"

Alpha Products Co. has developed a series of small, low cost data acquisition and control cards based on a bus called the "A-Bus". The A-Bus is easily generated by a 5380 SCSI controller IC. A scramble wired cable or small adapter card (available from Alpha Products) is all that is needed, to connect between a 5380 and one or more A-Bus cards.

A-Bus cards currently available from Alpha Products include: analog-to-digital converters, digital I/O, stepper motor controllers, relay outputs, optically isolated inputs, and prototype cards for custom interfaces. A five slot A-Bus motherboard is also available, and multiple motherboards can be daisy-chained, so quite a few A-Bus I/O cards can be connected to a single 5380 SCSI interface.

Table 3 gives the recommended signal mapping between the 5380's SCSI interface and the A-Bus backplane signals. Alpha Products offers a small adapter card which provides this interconnection, or you can wire a cable to do this yourself.

It is essential that you provide termination on the SCSI/A-Bus bus, since the 5380 has open collector outputs. If you don't, you will get unreliable results! However, the A-Bus devices are not designed to drive the 220/330 ohm pullup/pulldown termination normally used on the SCSI bus. Therefore, you must replace the SCSI bus termination networks with higher resistance terminators. For example, you might replace the pullup/pulldown networks with 1K pullup devices instead.

Table 3

SCSI/A-Bus Interface

A-Bus Signal	Pin	Function	SCSI Signal	Pin
+12 Volts	1	not used; no connect		
-12 Volts	2	not used; no connect		
GROUND	3	Signal Ground	GROUND	odd
+5 Volts	4	not used; no connect		
INTERRUPT	5	not used; no connect		
D0	7	Data in/out (LSB)	-DB0	2
D1	8	Data in/out	-DB1	4
D2	9	Data in/out	-DB2	6
D3	10	Data in/out	-DB3	8
D4	11	Data in/out	-DB4	10
D5	12	Data in/out	-DB5	12
D6	13	Data in/out	-DB6	14
D7	14	Data in/out (MSB)	-DB7	16
A0	15	Address (LSB)	-I/O	50
A1	16	Address	-C/D	46
A2	17	Address	-MSG	42
A3	18	Address (MSB)	-REQ	48
-IN	19	Read Strobe	-SEL	44
-OUT	20	Write Strobe	-BSY	36
ENABLE 0-3	21-24	not used; ground	GROUND	odd

Although there are not enough 5380 output signals to generate the four A-Bus "Enable" signals, the implementation shown in Table 3 is sufficient to select as many as sixteen A-Bus cards. If the Enable lines are required, you might consider pressing an unused parallel port (e.g. Centronics printer port) into service.

The following two software listings contain typical assembly language code which can be used to write to an Alpha products RE-140 relay output card, and read from an Alpha Products IN-141 optically isolated digital input card.

Interfacing to the Opto-22 "PAMUX" Bus

Another example of a simple I/O bus which can be easily synthesized by a 5380 SCSI controller is the Opto-22 "PAMUX" bus. Like the Alpha Products A-Bus, the Opto-22 PAMUX bus is a simple parallel I/O bus with data, address, and read and write control signals. Opto-22 offers an assortment of PAMUX analog and digital I/O mounting racks. Up to 16 PAMUX mounting racks can be daisy-chained on a single PAMUX ribbon cable bus, and each PAMUX mounting rack can hold up to 32 I/O input or output modules, resulting in up to 512 I/O points.

As with the Alpha Products A-Bus, all that is required to tie a 5380 SCSI bus to the PAMUX bus is a scramble-wired 50 conductor cable. A suggested wiring scheme is given in Table 4. The Opto-22 Model UCA3 "kludge card" can be used to make the bus-to-bus conversion, as described in Example 1.

Table 4

PAMUX/SCSI Interface

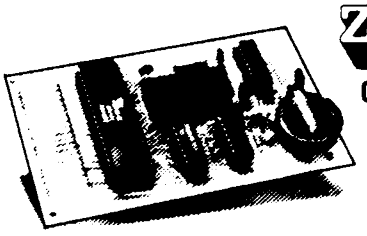
PAMUX Signal	Pin	Function	SCSI Signal	Pin
D0	47	Data in/out (LSB)	-DB0	2
D1	45	Data in/out	-DB1	4
D2	43	Data in/out	-DB2	6
D3	41	Data in/out	-DB3	8
D4	39	Data in/out	-DB4	10
D5	37	Data in/out	-DB5	12
D6	35	Data in/out	-DB6	14
D7	33	Data in/out (MSB)	-DB7	16
A0	1	Address (LSB)	-I/O	50
A1	3	Address	-C/D	46
A2	5	Address	-MSG	42
A3	7	Address (MSB)	-REQ	48
A4	9	not used, tie low	GROUND	odd
A5	11	not used, tie low	GROUND	odd
WRITE	13	Write Strobe	-BSY	36
READ	15	Read Strobe	-SEL	44
RST	49	not used, tie low	GROUND	odd
GROUND	even	Signal Ground	GROUND	odd

Although there are not enough 5380 output signals to generate all six PAMUX address signals, the implementation shown in Table 3 is sufficient to select up to 128 I/O points (32 I/O's on up to four PAMUX mounting racks). If more address lines are required, they may be able to be provided by an unused parallel port (e.g. Centronics printer port) into service. Be sure to ground A4 and A5 in your adapter cable (or on the UCA3).

The software routines needed to interface with the PAMUX modules are similar to those indicated in Example 3 for the A-Bus.

Here are a few differences from the A-Bus example:

(1) The PAMUX module bits can individually be inputs or outputs. As mentioned in Example 1, you can support a mixture of input and output modules in the same 8-bit group by writing 0's to the bits in the 5380's SCSI Data Register bits that are to be used as inputs so that those bits on the data bus are free to be driven by the PAMUX input modules.



Listing 1. Relay Output Card Interface

```
; **************************************************************************
; This is a demo of the Alpha products relay output card
; using their SCSI adapter.  The code is meant to run on
; an AMPRO Little Board/Z80 Z80-based single board system.
;
; Written 12/01/87 by Rick Lehrbaum
;
; Equates:
;
MODE$REG        EQU     22H     ;Bit 6 used to put in target mode.
TARGET$BIT      EQU     40H     ;Value to write to MODE
ADDRESS$REG     EQU     23H     ;Lower four bits used as A0-A3.  Inverted.
DATA$REG        EQU     20H     ;Eight bits of data.  Inverted.
CONTROL$REG     EQU     21H     ;Used to control data transfer, as follows:
        ; BIT  7  6  5  4  3  2  1  0
        ;                      |  |  L------ ASSERT DATA BUS when = 1
        ;                      |  L----------- READ STROBE when = 1
        ;                      L--------------- WRITE STROBE when = 1
; Based on these definitions, the read/write functions can use these values:
ASSERT$BIT      EQU     01H     ;Assert data bus.      Write to CONTROL.
WRITE$BIT       EQU     08H     ;Assert write strobe.  Write to CONTROL.
READ$BIT        EQU     04H     ;Assert read strobe.   Write to CONTROL.
;
                ORG 100H
;
INIT:
; Initialize the 5380 interface
                LXI     SP,1000H        ;Set stack pointer
                XRA     A
                OUT     CONTROL$REG     ;Disable all strobes and data bus
                MVI     A,TARGET$BIT
                OUT     MODE$REG        ;Place 5380 in target mode
                JMP     PROGRAM
;
WRITE:
; Writes the data byte in Register C to the I/O card at
; the address in Register B
                MOV     A,B             ;Get the address from B
                CMA                     ;Invert it
                OUT     ADDRESS$REG     ;Put the address on the bus
                MOV     A,C             ;Get the data from C
                CMA                     ;Invert it
                OUT     DATA$REG        ;Write it to the data port
                MVI     A,ASSERT$BIT
                OUT     CONTROL$REG     ;Assert the data bus
                MVI     A,ASSERT$BIT OR WRITE$BIT
                OUT     CONTROL$REG     ;Assert the write strobe
                MVI     A,ASSERT$BIT
                OUT     CONTROL$REG     ;Clear the write strobe
                XRA     A
                OUT     CONTROL$REG     ;Release the data bus
                RET
;
PROGRAM:
; This a simple sample program intended for the relay output card.
; It switches each relay on in sequence, and loops indefinitely.
                LXI     B,0001
LOOP:
                CALL    WRITE
                CALL    DELAY
                LXI     B,0002
                CALL    WRITE
                MOV     A,C
                RAL
                MOV     C,A
                JMP     LOOP            ; Loops forever
;
DELAY:
; Delays approximately 1 second.
DELO1:
                DCX     H
                MOV     A,H
                ORI     0
                JNZ     DELO1
                MOV     A,L
                ORI     0
                JNZ     DELO1
                RET
;
                END
; **************************************************************************
```

Listing 2. Opto-Isolated Input Card Interface

```
; **************************************************************************
; This is a demo of the Alpha products optically isolated input
; card using the Alpha Products SCSI adapter.  The code is meant to
; run on an AMPRO Little Board/Z80 Z80-based single board system.
;
; This demo must be run from DDT or used as a subroutine by another
; program.
;
; Written 12/03/87 by Rick Lehrbaum
;
; Equates:
;
MODE$REG        EQU     22H     ;Bit 6 used to put in target mode.
TARGET$BIT      EQU.    40H     ;Value to write to MODE
ADDRESS$REG     EQU     23H     ;Lower four bits used as A0-A3.  Inverted.
DATA$REG        EQU     20H     ;Eight bits of data.  Inverted.
CONTROL$REG     EQU     21H     ;Used to control data transfer, as follows:
        ; BIT  7  6  5  4  3  2  1  0
        ;                      |  |  L------ ASSERT DATA BUS when = 1
        ;                      |  L----------- READ STROBE when = 1
        ;                      L--------------- WRITE STROBE when = 1
; Based on these definitions, the read/write functions can use these values:
ASSERT$BIT      EQU     01H     ;Assert data bus.      Write to CONTROL.
WRITE$BIT       EQU     08H     ;Assert write strobe.  Write to CONTROL.
READ$BIT        EQU     04H     ;Assert read strobe.   Write to CONTROL.
;
                ORG 100H
;
INIT:
; Initialize the 5380 interface
                LXI     SP,1000H        ;Set stack pointer
                XRA     A
                OUT     CONTROL$REG     ;Disable all strobes and data bus
                MVI     A,TARGET$BIT
                OUT     MODE$REG        ;Place the 5380 in target mode
READ:
; Reads the data byte from the I/O card at the address in Register B
; and returns the data in register C
                MOV     A,B             ;Get the address from B
                CMA                     ;Invert it
                OUT     ADDRESS$REG     ;Put the address on the bus
                MVI     A,READ$BIT
                OUT     CONTROL$REG     ;Assert the read strobe
                IN      DATA$REG        ;Read the data
                CMA                     ;Invert it
                MOV     C,A
                XRA     A
                OUT     CONTROL$REG     ;Clear the read strobe
                RET
                END
; **************************************************************************
```

(2) The PAMUX bus is designed to be terminated with 180/390 ohm pullup/pulldown terminators. This is too heavy a termination for the 5380, so do not use the standard PAMUX "TERM1" terminator. Instead, use a standard SCSI termination (220/330 ohms) on at least one end—and preferably both ends—of the SCSI/PAMUX bus.

(3) All signals on the PAMUX bus are "active high", while all those of the SCSI bus are "active low". This means that everything must be inverted, including ADDRESS, DATA, and CONTROL SIGNALS. Consequently, the normal state of the SEL and BSY bits would need to be 1's, rather than 0's, in the 5380's Initiator Command Register. One or the other of those bits in the 5380 is then set to a 0 to generate a READ or WRITE strobe.

(4) Opto-22 recommends a 2 microsecond minimum duration for the WRITE strobe, and that you delay for at least 2 microseconds from the setting of the READ strobe prior to reading input data.

The SCSI/IOP Alternative

It is important to remember that all of the techniques of using the 5380 as a generalized I/O port discussed in this article assume that the 5380 is not going to be used for normal SCSI peripheral device connection as well. This means that if you plan to use a SCSI hard disk, tape drive, bubble drive, optical drive, RAM disk, or any other such SCSI device, you cannot also use the bus for simple digital I/O or to interface with an I/O bus such as the Alpha Products A-Bus or the Opto-22 PAMUX bus.

A unique device, available from AMPRO Computers Inc., does allow the SCSI bus to be used to add data acquisition and control devices along with normal SCSI devices. The SCSI/IOP is a card which acts like a "legal" SCSI target device, and implements a STD Bus device interface. A SCSI/IOP can be used with a single STD Bus I/O card, to add an individual function such as analog or digital I/O, or the SCSI/IOP can plug directly into an STD Bus backplane if multiple STD Bus I/O cards are required. The Z80A microprocessor on the SCSI/IOP can also be used to run tasks autonomously, so the SCSI/IOP can even add improved real time performance and multi-tasking to an non-real-time, and single-tasking disk operating systems such as PC-DOS and CP/M. ■

References

The following companies were mentioned in this article:

NCR Microelectronics Division
1635 Aeroplaza Drive
Colorado Springs, CO 80916
(303) 596-5612
(800) 525-2252

OPTO-22
15461 Springdale St.
Huntington Beach, CA 92649
(714) 891-5861
(800) 854-8851

ALPHA PRODUCTS CO.
242 West Avenue
Darien, CT 06820
(203) 656-1806

AMPRO COMPUTERS, INC.
1130 Mtn. View Alviso Rd.
Sunnyvale, CA 94089
(408) 734-2800

Computer Corner

(Continued from page 44)

that runs FORTH directly. The first production run is the NOVIX 4000 and I have bought an evaluation board from Chuck. There are plenty of these devices available now in several designs, the most popular are the PC compatible plug-ins. These boards have 512K of memory, up to 5MHZ operation, cross compilers for "C", use PCDOS for I/O, while screaming along at 5MIPS operation. Now that is 5MIPS at 5MHZ, running regular stuff, not some special program to show just how fast it works. If you check most of the high speed processors of late you will find most 5MIP units running at 15 to 20MHZ, and doing special register moves. The Novix can do several operations at once, but its speed comes from running FORTH directly.

That statement had me confused for a while, then I got more information and figured it all out. The Novix is like all CPUs in that it takes a bit pattern and converts that into commands. The difference here is that those bit patterns and the internal architecture directly correspond to FORTH. Forth is stack oriented and performs operations based on stack movements. The Novix makes use of all those design concepts by using bit-slice parts and gates. The device only has about 4000 transistors, while the 68020 and 80386 are into the half million devices. This unit is also CMOS which means it uses about 50MA of power (runs off of flashlight batteries).

If you are interested in NOVIX, I recommend one of the PC plug in boards, unless you are a real experimenter. Chuck's board is a bit high tech and not of my liking. He uses a European design, with push sockets and chips on both sides of the board. There are only 11 devices including the oscillator and the Novix. The board uses 6 static RAMs and 2 ROMs, as it does everything 16 bits at a time. That leaves one inverter and driver for the clock and reset line. The serial is handled by a resistor in the input line, while the output works using 5 volts. The Novix chip does everything, including bit banging the serial data at any baud rate, not bad for Forth.

I am still playing with the unit and will be getting some more software, and thinking about using it to drive disk drives directly. Chuck has a complete system running using only his board. That is driving video, reading and writing disk data, getting keyboard input, as well as running programs, all by adding only two or three more chips to the board I described. I was thinking about using it on an S-100 system, but have been giving more thought to making a PC size unit, that could talk to PC expansion boards.

What's Next

My main objective for the near future is finding a new job, but I haven't let that stand in my way of hacking. I am busy porting cheap operating systems to 68Ks and trying to get more time to work with Forth or Novix. In any case the number of projects seems to match or exceed the amount of time available...lets see didn't Murphy have a law covering that too? ■

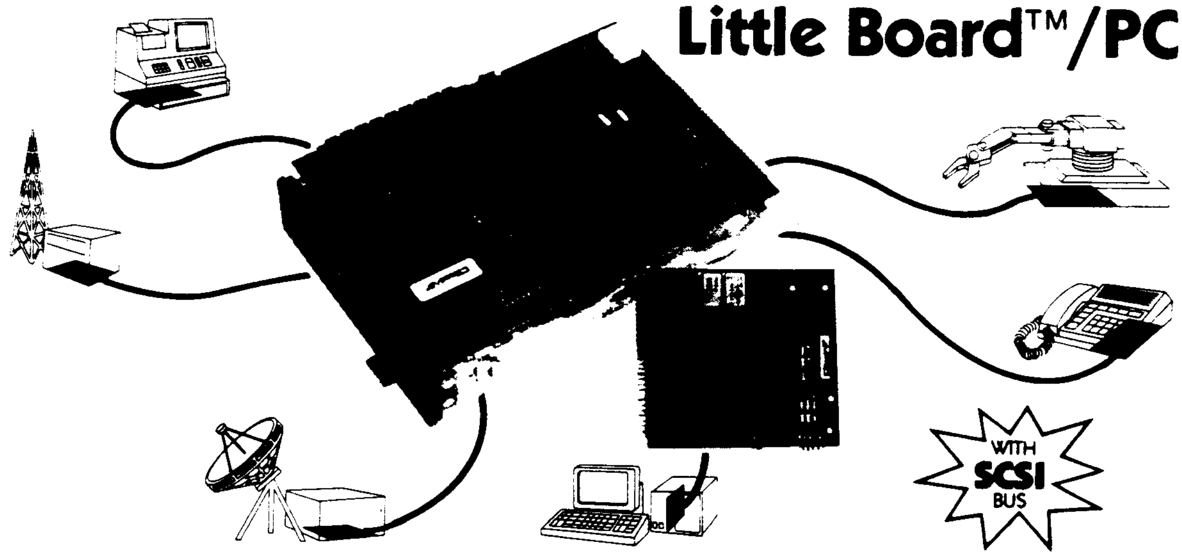
Compact, Low Power, Cost Effective
Single Board Computers
for Embedded Applications

World's smallest PC — and CMOS too!

A Motherboard and 4 Expansion Cards in the Space of a Half-Height 5-1/4″ Disk Drive!

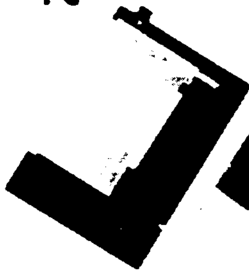
from **$359**
Qty 1

Little Board™/PC



WITH
SCSI
BUS

Stackplane™/ PC



Little Board/186

High performance single board MS-DOS system 8/16 MHz

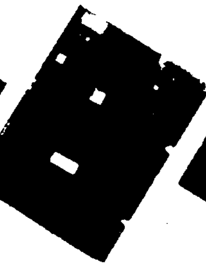
Expansion/ 186

Multi-function expansion for LB/186, I/O, Serial, RAM, Math, Clock

Project Board/186

Prototype adapter for 80186 based projects and products

Little Board/80



Least expensive single board CP/M system!

Project Board/80

Prototype adapter for Z80 projects and products

Distributors • **Argentina:** Factorial, S.A. 41-0018 • **Australia:** Current Solutions (613) 720-3298 • **Austria:** International Computer Applications GMBH 43-1/45 45 01-0* • **Brazil:** Computadores Compuleader (41) 262-4866 • **Canada:** Tri-M (604) 438-0028 • **Denmark:** Danbit (03) 66 20 20 • **Italy:** Microcom (6) 811-9406 • **Finland:** Symmetric OY 358-0-585-322 • **France:** Egal Plus (1) 4502-1800 • **Germany, West:** IST-Elektronik Vertribes GmbH 089-611-6151 • **Israel:** Alpha Terminals, Ltd. (03) 49-16-95 • **Spain:** Hardware & Software 204-2099 • **Sweden:** AB Akta (08) 54-20-20 • **Switzerland:** Thau Computer AG 41 1 740-41-05 • **UK:** Ambar Systems, Ltd. 0296 435511 • **USA:** Contact Ampro Computers Inc.

AMPRO
COMPUTERS, INCORPORATED

1130 Mountain View/Alviso Road
Sunnyvale, California 94089
(408) 734-2800
TLX 4940302 FAX (408) 734-2939

Communicating with Floppy Disks

Disk Parameters and Their Variations

by E. Stiltner, Skunk Creek Computing Services

Floppy disks have become a nearly universal medium of permanent data storage for microcomputers. To a smaller extent, they have become a medium of data interchange between microcomputers. But, with the exception of the "standard" SSSD 8 inch format, and some "universal" disk read programs, we are still living in a world where Brand A computer reads only Brand A diskettes, Brand B computer reads only Brand B diskettes, ditto for Brand C, and so on.

We have seen how the data from the computer is converted to a bit stream and stored on the diskette along with a very sophisticated set of control information in *Floppy Disk Track Structure* by Dr. Edwin Thall, in TCJ #29. Now, let's take a detailed look at the subject of floppy disks in terms of all the dimensions of actually recording the data.

The following parameters uniquely characterize the recording of computer data on a floppy disk:

Physical diskette size
Transmission rate
Encoding mode
Data true or inverted
Tracks density
Number of tracks per side
Number of sectors per track
Sector numbering
Physical data sector size
Number of sides
Sector skew

Physical diskette size

Floppy disks come in three physical sizes—8 inch, 5¼ inch, and 3½ inch. Since the 3½ inch diskettes are really 90 mm, they will be so referred for the rest of this article.

Transmission Rate

Date are transferred between a floppy disk drive and the computer at either of two transmission rates, 250 Khz or 500 Khz bits per second. All 8 inch drives use the 500 Khz rate. When the 5¼ inch drives first came along, a slower transmission rate of 250 Khz was used. Recent 5¼ inch drives support both transmission rates. The low rate was likewise adopted for the first generation of 90 mm disk drives. But recently introduced 90 mm drives support both low and high transmission rates.

Encoding Mode

There are two modes of encoding the data—i.e., transforming 8-bit bytes into a serial bit stream for the track image—Frequency Modulation (FM), and Modified Frequency Modulation (MFM). FM mode (sometimes called "low density") records two pulses per data bit, a clock pulse at the beginning of the data cell and pulse (or lack thereof) at the center of the data cell for the data. MFM mode (also called "high density") uses a self-clocking technique whereby a pulse appears at the center of the bit cell for a one, or at the beginning of the bit cell for a zero; a technique that effectively doubles the amount of data that is stored in a given space.

Data True or Inverted

In the world of electronic data, a datum can be represented in two forms—true or inverted. For example, a recording of the bit pattern 01100101 is just that. Or data can be represented in inverted form, where the above bit pattern is recorded as 10011010. Most floppy disk recording is in true mode. But there are diskettes with the data recorded in inverted form.

Track Density

The track density, i.e., the separation between the consecutive circles that make up the image of recorded data, has been very much a function of technology. The 8 inch drives recorded at 48 tracks per inch (tpi) or 1.9 tracks per millimeter. An 8 inch drive is an 8 inch drive and we don't worry about track density per se. It is of interest to note that the total width of the recording surface on the diskette is 40 mm wide.

But things changed with the 5¼ inch drives. The early 5¼ inch drives recorded the track images at the same 48 tpi but used a narrower recording surface width of 21 mm, which accounted for the small capacity of those units. The next technical development was 5¼ inch drives that record data at 96 tpi or 3.8 tracks per millimeter, effectively doubling the capacity. But this introduced another level of confusion for the computer user—now we are talking about "quad density." And the programmers talk about "double tracking" the 96 tpi drives to handle the 48 tpi diskettes which is why we need to be concerned with this parameter.

The next hardware generation was the 90 mm drives. These drives pack the track images at 130 tpi or 5.1 tracks per millimeter. The recording area on these diskettes is 15.6 mm wide.

Number of Tracks Per Side

How many tracks of data that are recorded on one side of a diskette depended on the technology. The 8 inch drive designers found that 77 tracks in a recording width of 40 mm was a reliable maximum.

With the 5¼ inch diskettes, the 48 tpi units recorded a total of 40 tracks in a recording width of 21 mm. The 96 tpi units record in the same width, so they have a total of 80 tracks per side.

The 90 mm drives provide the same characteristics as the 5¼ inch drives, a total of 80 tracks per side, but in a recording width of 15.6 mm.

Note that track numbers as seen by the computer range from 0...N-1; so for an 80 track diskette, track numbers 0...79 are available.

Sector Numbering

Tracks are assigned numbers ranging from 0...N – 1. This has been "built into the silicon." But sector numbers are another matter. Sector numbers are defined during the formatting of the diskette and can be any value supported by the hardware. In practice, most data are recorded with the sector numbers ranging from 1...N. But there are exceptions; some computer systems use a 0...N – 1 sector numbering assignment.

Physical Data Sector Size

Computer data are recorded on a diskette in standard length sequences called sectors. At present, sector lengths of 128, 256, 512, 1024 and sometimes 4096 bytes are supported. Note that the length of the physical sector size does not necessarily match the system or logical sector size. For example, the old CP/M system used a system sector size of 128 bytes, but various physical sector sizes (called "blocking") have been used by different manufacturers.

Keep in mind that the "real physical sector" image consists of the computer data surrounded by the control information described in Dr. Thall's paper.

Number of Sectors Per Track

The total number of bytes (this includes the control information overhead) that can be recorded on a track depends on the encoding mode and transmission rate. Figure 1 gives the typical number of bytes that can be stored on a track.

Figure 1

Mode	Low Rate	High rate	
FM:	3125	5210	Bytes per track
MFM:	6250	10420	Bytes per track

Given the total amount of data storage available on the track, the physical sector size, and the overhead for control information, we can calculate how many sectors of computer data can be stored on a track.

Table 1 presents typical sectors per track, based on 80 tracks per side and typical control information.

Table 1:

Sectors per Track

	Low Rate:		High Rate:		
	FM	MFM	FM	MFM	
128 byte sectors:	16	26	26	48	Sectors per track
256 byte sectors:	9	16	15	26	Sectors per track
512 byte sectors:	5	9	8	15	Sectors per track
1024 byte sectors:	2	5	4	8	Sectors per track

Table 2:

Floppy Diskette Capacity

	Low Rate:		High Rate:		
	FM	MFM	FM	MFM	
128 byte sectors:	160k	260k	260k	480k	Bytes per side
256 byte sectors:	180k	320k	300k	520k	Bytes per side
512 byte sectors:	200k	360k	320k	600k	Bytes per side
1024 byte sectors:	160k	400k	320k	640k	Bytes per side

("k" = 1024)

Number of Sides

The early disk drives supported recording data only on one side. Most state-of-the-art drives support both single or double sided recording. It is another datum to keep track of.

But there is a wrinkle to recording on the second side of the diskette that we must consider. What algorithm do we use for mapping the second side? We could do any of the following:

1) Extend the track number; e.g., the track numbers on the second side, as far as the operating system is concerned, range from 80...159.

2) Or we could extend the sector numbers for a track; e.g., the sector numbers on the second side, as far as the operating system is concerned, extend from the numbers on the first side. For example if there are 48 sectors per track, the sectors on the second side would be 49...96.

3) Or we could alternate—even track numbers on the first side, and odd track numbers on the second side.

In practice, all three schemes have been used by various manufacturers, so we have to keep track of which algorithm was used.

Sector Skew

Now we look at an efficiency consideration. Consider the physical situation upon reading data from the diskette: The computer system calculates the address of just what sector to read and issues the commands to read it. That sector is read. While the system or application program is digesting that sector, the diskette is still rotating. If the next sector to be read has just gone past the read head by the time the system issues the read for that sector, the computer has to wait for the diskette to rotate all the way around for the desired sector to come under the read head.

This problem is addressed by offsetting the sector numbers, either in the track image (hardware skew) or via the operating system (software skew).

For physical skew, the sector number in the sector ID field in the sector images is offset by the skew factor. For example, given a skew factor of 4, the first physical sector is number 1, the fifth physical sector is number 2, the ninth physical sector is number 3, and so on.

For software skew, the diskette is formatted with a hardware skew of one and the system uses a conversion table to map logical to physical sectors.

The goal is to have the next sector with the desired data come under the read head about the same time the system is ready to access it. A lot of effort goes into deciding just what is the optimum skew factor. And it can depend on the application; a high-level language can have a lot of overhead between successive sector reads and thus diskettes for that application need a large skew. Whereas a tight program in assembly language may get back to read its next sector fast enough to support a small skew factor.

Skew is also termed Interlace or Sector Translate.

Typical skew values range from 2 to 6.

Software Skew Versus Hardware Skew

The following are some considerations to apply in deciding between hardware and software skew

With hardware skew, the physical sector numbers are true; if the mapping algorithm says sector N, then sector N is the one to read. Memory space is not needed for the skew table to translate between logical and physical numbers. Diskettes with different hardware skew factors can be used without any system changes.

With software skew, the physical sector numbers are not true; the specific physical to logical mapping for that diskette is needed to read the data on another system. Memory space is needed for the skew table to map the logical system sec-

Appendix I

```
                    Floppy Diskette Parameters

Date:  / /                          Computer:____________________

Physical size:
   3) 3 1/2 inch (90 mm)
         or
   5) 5 1/4 inch
         or
   8) 8 inch......................................._______

Transmission rate:
   L) Low, 250 Kilobits/sec with MFM
         or
   H) High, 500 Kilobits/sec with MFM................_______

Encoding mode:
   F) FM, Frequency Modulation
         or
   M) MFM, Modified Frequency Modulation..............._______

Data true or inverted:
   T) Data true
         or
   I) Data inverted..................................._______

Track density:
   A) 48 tracks/inch (8 inch drives)
         or
   B) 48 tracks/inch (5 1/4 inch drives)
         or
   C) 96 tracks/inch (5 1/4 inch drives)
         or
   D) 130 tracks/inch (3 1/2 inch drives)............._______

Number of tracks per side:
   A) 77 tracks (8 inch drives)
         or
   B) 40 tracks (5 1/4 inch drives)
         or
   C) 80 tracks (5 1/4 inch and 3 1/2 inch drives)....._______

Physical sector size:
   1) 128 bytes per sector
         or
   2) 256 bytes per sector
         or
   3) 512 bytes per sector
         or
   4) 1024 bytes per sector..........................._______

Number of physical sectors per track:
      5 ... 52........................................_______

Sector numbering:
   A) 0 ... N-1
         or
   B) 1 ... N........................................._______

Number of sides:
   1) 1 side
         or
   2) 2 sides........................................._______

Double sided mapping:
   A) Extend Sectors
         or
   B) Extend Tracks
         or
   C) Alternate Sides................................._______

Hardware skew:
      1 ... N........................................._______

Software skew:
      1 ... N........................................._______
```

Appendix II

There is a lot of changing back and forth between English and metric units in this area. The following table gives measurements that have been used in this paper in both systems.

Disk size:	
8 inches	203 millimeters
5 1/4 inches	133 millimeters
3 1/2 inches	90 millimeters
Track density:	
48 tracks per inch	1.9 tracks per millimeter
96 tracks per inch	3.8 tracks per millimeter
130 tracks per inch	5.1 tracks per millimeter
Recording surface width:	
1.6 inches	40 millimeters
0.83 inches	21 millimeters
0.62 inches	15.6 millimeters

tor to the physical sector. An advantage of software skew is that the formatting program can be simpler and does not need to be changed for different skew factors.

Needless to say, one should not mix hardware and software skews. That is, if the hardware skew is not one, then a software skew of one is expected and vice versa.

Putting it Together

Now that we have seen the relationships between the various parameters of recording data on floppy disks, we can calculate typical disk capacity. Table 2 gives a total disk capacity for a mythical 80-track diskette as a function of the major parameters of transmission rate, encoding mode, and sector size.

We have seen the nearly one dozen parameters that specify how computer data are recorded on floppy disks. Given reasonable permutations in each parameter, we get some 12,000 possible variations in the way data are recorded! Is it any wonder we have to contend with a few incompatibilities? Or perhaps we really should ask how many possible formats have NOT been used.

In hopes of minimizing further electronic misunderstandings, the log sheet in Appendix I is recommended to uniquely document just how the data on a particular diskette has been recorded.

In closing, note that each computer system overlays this elaborate structure with another layer of organization to support the system image, file directories, various state tables, and the actual file data. But that is another very long story and unique to just about each computer system. ■

XBOIS

A Replacement BIOS for the SB180

by Richard Jacobson

Xsystems Software's XBIOS is a replacement BIOS for Micromint's SB180 and SB180FX single-board computers which are based on the Hitachi HD64180 CPU. When the SB180 was first announced two years ago, the 8-bit world greeted the new system with enthusiasm and many people were convinced that Micromint's new computer represented the salvation of the 8-bit community. While the implementation of the 64180 by Micromint was quite decent on the operating system level, there were a number of irritations that plagued users, particularly those who had implemented hard drive systems using the COMM180 SCSI board and its associated BIOS. These irritations included slow disk I/O and, more significantly for many, extremely low TPA (many hard drive systems were running with TPA's less than the Digital Research standard of 48k.) Users with little technical software background were daunted by the perceived complexity of modifying the Micromint system. With the availability of XBIOS, such irritations are now a thing of the past.

XBIOS offers a dramatic improvement over the standard Micromint operating system in the critical areas of performance, configurability, and TPA. It will run on any configuration of the SB180 (with or without the Micromint SCSI interface) or the SB180FX and includes support for the ETS180IO+ add-on board, with which it is bundled.

A simple list of the features of XBIOS, though impressive, cannot do justice to the experience of using it, particularly for users who have grown accustomed to the limitations of the standard Micromint system.

By putting the BIOS in a bank of memory separate from the user's, XBIOS provides 3k to 5k more TPA than Micromint's system. The original, standard-issue operating system, while functional, has such stringent TPA limitations, particularly on hard disk systems, that many SB180's simply could not reasonably handle TPA-hungry programs such as WordStar Release 4® . In contrast, with XBIOS installed, a 20 meg hard drive system with all ZCPR3 segments except IOP, weighs in at an impressive 54k in the TPA department (the measurement includes the standard 2k for the CCP). An XBIOS-based SB180 will support full-featured WordStar Release 4 with ease—and with room to spare.

Not only does XBIOS offer significant improvements in TPA for all configurations of the SB180, but it also implements RSX-type programs, called banked system extensions (BSX's), which further extend functionality, without reducing TPA, by tucking useful programs into the alternate memory bank. DateStamper, by Plu•Perfect Systems, is supplied as a BSX. Other BSX's support CP/M 3.0-style date calls, using BDOS function 105. Users can anticipate a proliferation of BSX's as time goes by and as XBIOS users start implementing BSX's, limited only by their needs. For example, an HP-style calculator has already been implemented as a BSX. Numerous additional BSX's are planned by Xsystems Software, each modular BSX supporting various specialized hardware and software applications. The BSX system extensions can be loaded into and removed from the alternate bank of memory on the fly and are powerful enhancements to system flexibility. In short, a BSX offers all the advantages of a memory resident program without requiring the sacrifice of precious TPA or a needed ZCPR3 system segment. (Many useful programs are implemented as ZCPR3 RCP's but require the user to overwrite the standard "utility" RCP and therefore entail an overall loss of functionality as well as the expenditure of CPU time in loading different packages.) Concepts such as BSX's reveal the depth of systems-level thinking that has gone into XBIOS.

The execution of XBIOS reflects its author's fanatic attention to detail. The XBIOS package includes an integrated system clock/calendar to support time and date functions. It also includes a utility that allows any port to be used for CP/M I/O devices (LST:, RDR:, PUN:), and allows on-the-fly I/O redirection. XBIOS permits the console to be assigned to any port. The package supports full Xon/Xoff handshaking, as well as DTR for ETS180IO+ ports. XBIOS supports multiple controller and hard drives, using the standard SCSI interface. One of the hallmarks of XBIOS is flexibility—in actual use as well as in overall system configuration.

The flexibility of system configuration is in keeping with the emphasis of XBIOS on functionality. Each of the 16 logical drives can be assigned to any type—RAM disk, hard drive, or floppy. The XBIOS system treats the SB180's RAM disk as a single drive, with contiguous space, even on the FX version of the board. Even so, the user can partition the RAM disk into multiple logical drives. Furthermore, the "A" drive can be assigned to any device (no more need for system tracks), and the RAM disk can be assigned to any logical drive, not just "M", another limitation of the Micromint system. All of this flexibility can be taken advantage of by the average user. The installation of XBIOS does not require any knowledge of assembly language programming.

A powerful, menu-driven configuration utility makes installation a snap, allowing the user to define all system parameters, including the assignment of different physical parameters to each floppy disk drive, the assignment of each logical drive ("A" through "P"), and the sizing of Z-System buffers, as well as the choice of certain Z-System parameters.

The package includes full utility support: a floppy and hard drive formatter, a set-and-display-time utility, a program to reassign I/O devices, and a utility to initialize the RAM disk. All this comes with a beautifully printed and exhaustively detailed manual.

The ease of setup—as well as the extensive and thorough documentation—does not mean the user is prevented from getting into trouble. Power and flexibility always come at a price. This is not meant as a criticism of XBIOS package—it is as nearly perfect a piece of software as I have experienced and alone justifies purchase of the SB180 computer.

As an example of what can happen when you fail to pay appropriate attention to the option settings, or simply forget about what you have done entirely, I have a cautionary tale. About two weeks ago I added a 96 TPI drive to my SB180 system, configured XBIOS for three floppies instead of two, checked everything out, flipped the switch, and waited to enjoy the fruits of my labors. Nothing! The system would not boot. All the terminal showed was a partial sign-on message and then—out to lunch. Nothing worked. Finally, after horrible gyrations, I figured out the new system would not boot because it needed to be set for one memory wait state. Before the addition of the third floppy, and thanks to the flexibility of XBIOS, I had been running at zero memory wait states—a distinct speed advantage over the standard one-wait-state Micromint system. (The SB180 hardware always has one op-code wait state and the standard software adds another memory wait state. Taking out the standard single memory wait state under XBIOS gives about a 20% increase in speed and makes the 6mhz SB180 almost as fast as the 9mhz version.) Apparently, the slight increase in power draw caused by the added floppy drive was the straw that broke the camel's back. Sure enough, when I configured XBIOS for one memory wait state the system booted like a charm. As I said, power comes at a price.

The price is no price at all when it comes to XBIOS. The increase in TPA alone makes the package a bargain. The operating system addresses, ZCPR3.3 system segments, and ZCPR3.3 buffers on my XBIOS fueled SB180 prove my point and are shown below:

```
ZCPR3 Element    Base Address
-------------    ------------
     CCP            D000 H
     8DOS           D806 H
     BIOS           E600 H

Env Descriptor      FE00 H

Pack: FCP           FA00 H
       IOP          0000 H
       RCP          F200 H

Buf: Cmd Line       FF00 H
     Ext FCB        FDD0 H
     Ext Path       FDF4 H
     Ext Stk        FFD0 H
     Messages       FD80 H
     Named Dir      FC00 H
     Shell Stk      FD00 H
     Wheel Byte     FDFF H
```

This setup lacks only an IOP, which I do not take frequent enough advantage of to warrant the TPA loss. The bottom line is that the memory map above buys me a 54k TPA, with a full-up Z-System except IOP, and that's good for any piece of hardware.

Two years ago, the author of XBIOS, Malcom Kemp, purchased the SB180, which had just been announced in Byte magazine. (Previously, he had owned a Morrow floppy-based system.) Kemp loved Echelon's Z-System, but was frustrated by the relative lack of TPA on the SB180, as well as the slow disk I/O. That frustration was the beginning of XBIOS.

Kemp decided to put a portion of the BIOS into an alternate bank of memory. (The Hitachi 64180 chip has memory management that allows logical remapping of where the 64k addressable bytes exist in physical memory, permitting control over a total of 256k of memory on the chip. The FX version of the SB180 uses an enhanced MMU, allowing control over more memory.) At first, Kemp put only the disk I/O routines and the disk buffers in the alternate bank. The current release of XBIOS has the entire BIOS in the alternate bank (carved out of the SB180's RAM disk—so beware of a small reduction in RAM disk size) except for a small transfer vector, a short front end to the interrupt handlers, a routine to save machine state, and disk tables along with their associated buffers.

During his initial work on XBIOS, Kemp exchanged ideas with Jay Sage, Bridger Mitchell, Bruce Morgen, Ken Taschner, and others. The BSX concept, as well as others, grew out of Kemp's creativity and his interaction with these systems programmers. At all times, however, Kemp's main effort has been to maximize the configurability of the SB180. As Kemp says, "Whatever the hardware could support, I wanted to support with software." The I/O redirection capabilities of XBIOS grew out of this concept.

Not only does XBIOS support the hardware capabilities of the SB180, but it also supports the greater functionality of the ETS180IO+ board, by Electronic Technical Services, with its additional serial ports. The ETS180IO+ board, carried by all XBIOS distributors, is a vast improvement over the standard SB180 hardware I/O facilities. The board is an all CMOS design, offering two additional high-speed (115.2 kbps) serial ports with CPU independent baud rates and full hardware handshaking. It includes 24 bits of user configurable parallel I/O. In addition, there is full buffering of the expansion bus for other add-on boards, as well as comprehensive interrupt support. The SASI/SCSI interface is fully DMA capable and, when combined with XBIOS, supports all popular SCSI-compatible hard disk controllers and drives. The battery-backed, real-time clock, along with DateStamper time and date stamping of disk files, is a real improvement over standard kludges. The real-time clock is accurate to 10 ppm, supporting month, day, year, day-of-week, hours, minutes, and seconds. It uses standard BR2325 lithium batteries for a back-up power source. The ETS180IO+ board is 7.70 inches long by 4.12 inches wide. The power required is 300 MW typical, 500 MW maximum, and excludes SCSI. Both RS232 ports are factory configured as DCE, but each may be independently jumpered to DTE.

Needless to say, while XBIOS functions happily without the ETS180IO+ board, the addition of that board provides a significant increase in power and flexibility to any SB180-based system.

Malcom Kemp plans a Release 2.0 of XBIOS. He hopes to concentrate on speeding up both disk and character I/O (perfectly fine in Release 1.0, but Kemp is never satisfied). There was an early Release 0.8, which did not support the FX board, but which did have slightly faster disk and character I/O than the current release. XBIOS now fully supports all SB180's, in all configurations.

Criticisms? Nothing is perfect and XBIOS has some features currently missing or some areas where improvement is possible. While this is not really a problem, character I/O is slower than on the standard Micromint system. Also, Uniform will not run under XBIOS. Both of these criticisms will be answered in future XBIOS developments. Malcom Kemp is committed to making the finest operating system available for the SB180 even finer.

XBIOS is available from the following sources: (1) Lillipute Z-Node, 1709 N. North Park Avenue, Chicago, IL 60614, Modem:312-649-1730, 312-664-1730, Voice:312-280-1621; (2) Sage Microsystems East, 1435 Centre Street, Newton, MA 02159, Modem:617-965-7259, Voice:617-965-3552; and (3) NAOG/Z-SIG, P.O. Box 2871, Warminster, PA 18974, Voice:215-443-9031. The price is $75 plus S & H. All of these vendors carry the ETS180IO+ board as well. ■

K-OS ONE and the SAGE
Demystifing Operating Systems
by Bill Kibler

The people at Hawthorne Technology have put together an inexpensive, but efficient operating system for 68000 computers, K-OS ONE® . The original design concept was for an inexpensive system, in which all the code was provided, so that hackers could still do something on their own. We find that most systems today have become so complex that it is impossible, in many cases, to get to the hardware directly. These companies in fact have gone out of their way to make sure that the user can not change or modify their system in any way. Now that is fine if all you want to do is run commercial packages of software.

If your desires run to making a system to protect your home, or to talk to people when you are not around, a non standard design might be more to your liking. If you are just starting to get into hardware design and want to run special programs to test out that design, multi-layers of operating system are not what you want. All these design considerations require the operating system to be simple and straight forward. The installation should be easy and provide for many options or levels of development.

All of these design considerations were behind the development of the K-OS ONE operating system. We felt that the 68K was superior to the more common CPUs in use today, but the lack of an inexpensive operating system was preventing people from discovering its features. Like any project, this one has some learning and work attached to it. Most people find operating systems a mystical concept, and feel that writing operating system programs is beyond their capbilities. What I hope to do here is demystify the topic, especially the installation of K-OS ONE.

Getting Started

The major stumbling block for most people is just deciding where to start. It took me several days of looking at various things before I could chose a direction to go. The first thing needed is a computer system. If the system is already running so much the better. If the system is not running, special problems must be handled first. What I am going to cover here is bringing up K-OS on a Sage/Stride computer. At a later time I will expand on getting a system up from scratch for the first time. What we are interested in here now is what steps are needed and what you will need to learn to get the job done.

The first place to start is learning about your current system. The Sage is a 68K based unit, mine is five years old. The unit came with all the books and software including source code for all the current PROMS AND BIOSs. Without the source code it is almost impossible to bring up older systems. It is possible with just schematics to figure out how everything talks to each other, but looking at all the older programs, will make some of the items quickly clear. I printed out all the Sage source code, about two inches worth, which is what most complex system will be—very long.

To help us understand how to start, we need to review how, and what steps, occur in getting the operating system running. The hardware on reset goes to a PROM which must contain a program to start the system. This is called a BOOT program. The boot program will initialize the system enough to start some form of operation. The better systems also contain a DEBUGGER or MONITOR, should some problem or special action be needed to bring it up. In the Sage, the PROM reads some switches on the back and determines which actions to take. Normally it will test memory, then boot the system. Options are to not test memory, and go to debugger. In the debugger, a simple command will start the system, or you can disassemble the memory.

After the reset, we have a number of functions that must be performed, such as initializing the I/O devices. The initial setting of the baud rate for your serial terminal is taken from the switch setting in the Sage. The parallel devices also need to know which lines are to be input and which to be output. The disk drives should be reset to track zero and maybe even checked as to what type they are. These are the typical actions that occur after reset. If you enter into the debugger, at this point you can explore your system or do a "IF" in the Sage which starts the booting action from a floppy drive. At that point this system goes and does its boot action which means loading a BOOTSTRAP program at a fixed location and jumping to it.

Each operating system will have its own disk format and number of files which must be loaded in order to bring up the system. Most operating systems are broken into three parts; BIOS, BDOS, and COMMAND. The BIOS stands for Basic Input Output System, and does all the talking to the hardware directly. The BDOS is your Basic Disk Operating System and provides a uniform means of having programs talk to different forms of hardware. The programs will make calls to the BDOS and it will convert them into the required number of commands needed to achieve the task requested. Typically you may have a terminal and a printer installed. By sending the proper command you can ECHO all output to your terminal to the printer. The BDOS handles the echo-ing while the BIOS actually makes separate outputs to the terminal and the printer, each being a different routine in the BIOS.

The COMMAND processor takes keyboard input and interprets it into a number of predefined operations, such as displaying a directory of the disk. To display that directory it must request the BDOS to read the disk for the directory information, format that information and then send it to the terminal port through the BIOS via the BDOS. When running programs, it is typical to replace the command processor with your program, and then reload the command processor after your program ends. That operation is called warm booting.

In the K-OS those programs are SYSTEM.BIO for the BIOS, OPERATE.BIN for the BDOS, and

COMMAND.BIN for the command processor. In the 68000 the components can talk to each other by using regular jump tables and interrupt, or trap vectors. K-OS uses both tables and vectors. To bring the system up you will need to set values for both items, but then we are getting ahead of ourselves a bit here.

Bootstrapping

We that said after reset the system can automatically boot from disk or you can do this manually. In either case the Sage process is the same, two sectors are loaded from disk into memory location 400hex and then the system jumps to it. This is typical of all boot operations, what is different is the number of sectors, location, and a special Sage signature. It is at this point that we now get out our books and determine the format of our disk. The IBM PC line of disks use a 40 track format of 512 bytes per sector and are 9 sectors per track. The PC can read, and did use, other formats, but this is now the most common format. The next bit of information we need is the location of the directory information. The directory, or the information that tells you where the files are stored, is contained in two sections, FATs and directory entries. The PCDOS system is based on the original CP/M operating system which, only had 32 bytes set aside for each entry in the directory. In CP/M, the sectors that a file used were placed with the file name, which limited it to a 16K file size before another directory entry was needed.

The PC designers wanted to add date and time, as well as to allow larger files, so their answer was using File Allocation Tables or FATs. These tables tell the operating system which sectors were used, based on a starting pointer contained in the directory entry. Typically the FATs are sectors 2 through 5 with the directory entries being sectors 6 through 12. With each side containing only 9 sectors, directory entries 10, 11, and 12 are on side 1 (the sides are 0 and 1). The bootstrap PC loader is on sector 1 only and contains data other than the bootstrap. The book I used for most of the PC information is *Peter Norton's Programmer's Guide to the IBM PC*, and I can recommend it for more details.

You need to know this information, because K-OS uses the PC disk format. Without this compatibility the porting over of the system would be considerably more complex and time consuming. All things are not totally simple however, as the Sage is not PC compatible. What we

Boot Loader Listing

```
;
;       KOSONE BOOTLOADER ROUTINE - PC COMPATIBLE
;
;       Boot loader routine for the Sage/Stride computer,
;         written Sept 1987 by Bill Kibler some portions
;         supplied from HTPL sample BIOS: BIOSAMPL.ASM.
;
;       The Sage computer loads sector 1 and 2 when given
;       a boot command "IF". Each sector is 512 bytes long.
;       The first four bytes of the boot sector must contain
;       the word "BOOT" or the boot loader in the PROM will
;       error out. The code is loaded at 0400hex and the program
;       will jump to 0404h after checking for "BOOT". Also the
;       sectors 1 and 2 are logical blocks, 0 and 1 (1). The
;       Sage PROM reads sectors by logical block numbers not
;       track, side, and sector.
;
;       The PC disk format is compatible as far as sector and
;       track utilization. The PC uses only the first sector for
;       boot loader with the FATs starting at sector two. PCs use
;       Clusters of two sectors to a block while the Sage is
;       one sector per block for the floppy disks. This will
;       require doubling of the cluster number and subing one
;       to get the proper sector number for passing to the
;       Sage FDREAD routines.
;
;       The program loads ONLY the FIRST FAT and FIRST DIR
;       sector, to save space. This means that the SYSTEM.BIO
;       or BIOS file must be loaded within the first 15 files!
;       The BIOS code is loaded at A00h, space between 400h
;       and A00h can be used after BIOS is completely loaded.

;
        TITLE   "SAGE BOOT.ASM PC COMPATIBLE BOOTSTRAP LOADER"
********************** ORIGINS ******************************
BOOT            EQU     00000004H
BOOT_CODE       EQU     00000400H       ;BOOTER LOCATION
BOOT_VAR        EQU     000005E0H       ;SCRATCH AREA
BOOT_FAT        EQU     00000600H       ;FAT READ WITH BOOT
BOOT_DIR        EQU     00000800H       ;DIR 1 LOCATION
BIOS_CODE       EQU     00000A00H       ;BIOS CODE
TERMTEXT        EQU     00FE0018H       ;PRINTOUT TEXT STRING
TERMCRLF        EQU     00FE001CH       ;PRINTOUT CRLF
FDREAD          EQU     00FE0028H       ;READ FLOPPY DISKETTE
TOTRACK         EQU     40              ;TRACKS PER DISK
TOTSECT         EQU     9               ;SECTORS PER DISK
TOTSIDE         EQU     2               ;SIDES PER DISK
DEBUG           EQU     00FE0010H       ;DEBUG ENTRY
*************** SYSTEM INITIALIZATION ***********************
        ORG     BOOT_CODE
;
        DC.B    "BOOT"  ; Sage checks for this statement
;
;       PROM starts program here...
;
        LEA     BOOT_VAR,A3
        MOVE.L  (A7)+,RTNADD(A3)
        MOVE.W  (A7)+,DRIVE(A3)
;
        JSR     TERMCRLF        ;flag to booting is going on
        LEA     INITMSG,A0
        JSR     TERMTEXT
        JSR     TERMCRLF
;
        MOVE.W  #5,-(A7)        ;load first DIR sector
        LEA     BOOT_DIR,A0     ;DIR load location
        MOVE.L  A0,DIRPN(A3)    ;ALSO LOAD DIRPOINTER
        MOVE.L  A0,-(A7)        ;PUSH location on stack
        MOVEA.W #512,A0         ;one sector load
        MOVE.L  A0,-(A7)        ;PUSH sector length
        MOVE.W  DRIVE(A3),-(A7) ;PUSH drive number
        JSR     FDREAD          ;go read sector

        BNE.S   ABORT           ;go debugger if error

;
;------------------------------ LOAD BIOS

        LEA     BOOT_VAR,A3     ;setup pointer
        MOVE.L  #SYSNAM,FNAMPN(A3)      ;load string pointer
        MOVE.L  #BIOS_CODE,LOADPN(A3)   ;LOAD BIOS ADDRS
        BSR     FINDFIL
        BNE.S   ABORT           ;go debugger
        LEA     MSG1,A0
        JSR     TERMTEXT
        JSR     TERMCRLF
        BSR     LOADFIL
        BNE.S   ABORT
```

learn here is that the Sage loads sectors 1 and 2 as the bootstrap program. Sector 2 however is the first FAT and cannot contain boot program. This leaves 512 bytes for the program, less four bytes for "BOOT". Sage not only loads the program, but then checks to see if it is the correct program. The PROM reads the first four bytes looking for "BOOT", if not found it will abort to the debugger. When found it jumps then to 404hex (just pass "BOOT") and starts the BOOTSTRAP. The bootstrap must then load the BIOS, jump to it, and then the BIOS loads BDOS and command programs.

The Real Work

The real work involves getting enough information and program samples from the Sage BOOT loader, PROM, and BIOS listings to figure out how to load the BIOS. What must also be considered is handling the FATs and DIR data as they are in INTEL hex format. The 68000 stores address or data in memory with high values followed by low values. The Intel processors store the same information in LOW then HIGH, or backwards from real life (this is one reason people like Motorola products). Not only are values in the directory stored LOW then HIGH but the FAT table has 12 bit values with the bits shifted around. It is a bit funny, so just get a book and read about it. The answers to our problem are found in the sample BIOSs supplied by Joe Bartel who wrote K-OS. These sample BIOSs show just how to manipulate the Intel bits and FATs with 68000 assembly language.

There are several ways we can boot load the BIOS. If code length was no problem, we would load all the FATs and directories, shuffle through them till we found our program, and then load them. Space being limited, I decided to cheat a bit. I let the PROM load not only the boot program, but also the first FAT. I followed that by loading the first directory sector. Next I checked those two sectors for the file and its FATs, loading same. This requires that the BIOS be loaded first, before any other programs. You can load it several times and even a few others (not more than 16), but I would experiment with a freshly formatted disk and only the three files first.

The next question is how do I get them on the disk, especially the boot loader. The PCDOS comes with DEBUG as a utility for reading disk data as well as checking memory. I would look most of the commands up in the manuals first so you understand what you are doing. This

```
;
;**************** SET UP DONE, START A PROGRAM ************************
        JMP     BIOS_CODE       ;START BIOS
;
;**************** COULD NOT LOAD SYSTEM *********************************
ABORT   LEA     MSG2,A0
        JSR     TERMTEXT
        JSR     TERMCRLF
        JMP     DEBUG           ;EXIT TO DEBUGGER
        RTS
;
********************** SYSTEM LOAD ROUTINES *************************
;-------------------------------- FIND FILENAME IN DIRECTORY
FINDFIL
        MOVEQ   #15,D0
FIND20  MOVEQ   #10,D1
        MOVEA.L DIRPN(A3),A0
        MOVEA.L FNAMPN(A3),A1
FIND30  CMPM.B  (A0)+,(A1)+     ;COMPARE DIR ENTRY TO FILE NAME
        DBNE    D1,FIND30
        BNE.S   FIND40
        RTS                     ;RETURN TRUE IF EQUAL
FIND40  ADDI.L  #32,DIRPN(A3)
        DBRA    D0,FIND20               ;ENDFOR
        MOVEQ   #1,D0
        RTS                     ;RETURN FALSE IF NOT FOUND
;-------------------------------- LOAD BINARY FILE INTO MEMORY
LOADFIL
        MOVEA.L DIRPN(A3),A0    ;GET DIR.START
        ADDA.L  #26,A0
        BSR     LDINTELWORD
        BSR     BLKTOREC        ;CONVERT START BLOCK TO START RECORD
        MOVE.L  D0,RECORD(A3)
        MOVEA.L DIRPN(A3),A0    ;GET DIR.SIZE
        ADDA.L  #28,A0
        BSR     LDINTELLONG
        ADDI.L  #511,D0         ;CALC NUMBER OF RECORDS TO LOAD
        MOVEQ   #9,D1
        LSR.L   D1,D0
        MOVE.L  D0,RECCNT(A3)   ;FOR ALL RECORDS IN FILE
LOAD20  BSR     TRANSFORM       ;PUSH NEXT SECTOR/BLOCK

        MOVE.W  SECTOR(A3),-(A7) ;PUSH sector number
        MOVEA.L LOADPN(A3),A0
        MOVE.L  A0,-(A7)        ;PUSH location on stack
        MOVEA.W #512,A0         ;one sector load
        MOVE.L  A0,-(A7)        ;PUSH sector length
        MOVE.W  DRIVE(A3),-(A7) ;PUSH drive number
        JSR     FDREAD

        ADDI.L  #512,LOADPN(A3) ;ADVANCE POINTER
        BSR     NEXTREC         ;CALC NEXT RECORD NUMBER USING FAT
        SUBQ.L  #1,RECCNT(A3)
        BNE.S   LOAD20          ;ENDFOR
        RTS

;-------------------------------- CALC NEXT RECORD USING FAT
NEXTREC MOVE.L  RECORD(A3),D0
        BTST.L  #0,D0           ;IF ODD( RECORD ) THEN
        BEQ.S   NXRC10
        BSR     RECTOBLK        ;RECORD=BLKTOREC( NEXTBLK( RECTOBLK( RECORD)))
        BSR     NEXTBLK
        BSR     BLKTOREC
        BRA.S   NXRC20
NXRC10  ADDQ.L  #1,D0           ;ELSE RECORD = RECORD +1
NXRC20  MOVE.L  D0,RECORD(A3)
        RTS
;-------------------------------- CONVERT RECORD NUMBER TO FAT INDEX
RECTOBLK
        SUBI.L  #12,D0
        LSR.L   #1,D0
        ADDQ.L  #2,D0
        RTS
;-------------------------------- CONVERT FAT INDEX TO RECORD NUMBER
BLKTOREC
        SUBQ.L  #2,D0
        LSL.L   #1,D0
        ADDI.L  #12,D0
        RTS
;-------------------------------- GET NEXT BLOCK IN FAT CHAIN
NEXTBLK MOVE.L  D0,D1
        ADD.L   D1,D0   ;TABLEPOINTER=BLOCK*3/2+FATBUF
        ADD.L   D1,D0
        LSR.L   #1,D0
        LEA     BOOT_FAT,A0
        ADDA.L  D0,A0
        BSR     LDINTELWORD
        BTST.L  #0,D1   ;IF PREVIOUS WAS ODD
        BEQ.S   NXBL10
        LSR.L   #4,D0   ;THEN SHIFT OUT LOW NIBBLE
        RTS
```

```
NXBL10  ANDI.W  #0FFFH,D0;ELSE MASK OFF HIGH NIBBLE
        RTS
;-------------------------------- LOAD A WORD STORED IN INTEL FORMAT
LDINTELWORD
        MOVEQ   #0,D0
        MOVE.B  1(A0),D0
        LSL.L   #8,D0
        MOVE.B  (A0),D0
        RTS
;-------------------------------- LOAD A LONG STORED IN INTEL FORMAT
LDINTELLONG
        MOVEQ   #0,D0
        ADDQ.L  #4,A0
        MOVE.B  -(A0),D0
        LSL.L   #8,D0
        MOVE.B  -(A0),D0
        LSL.L   #8,D0
        MOVE.B  -(A0),D0
        LSL.L   #8,D0
        MOVE.B  -(A0),D0
        RTS

;--------------- CONVERT LOGICAL RECORD TO SECTOR/SAGE BLOCK
TRANSFORM
        MOVE.L  RECORD(A3),D0
        MOVE.W  D0,SECTOR(A3)
        RTS
;

**************** RUN TIME CONSTANTS *********************************
SYSNAM  DC.B    "SYSTEM  BIO",0
;
INITMSG DC.B    "HTPL-SAGE BOOTSTRAP",0
MSG1    DC.B    "LOADING BIOS",0
MSG2    DC.B    "READ ERROR",0
;
******************** VARIABLES **************************
        ORG     BOOT_VAR
;
VARS    EQU     $

;
RECORD  EQU     $-VARS
        DS.L    1        ;LOGICAL RECORD TO READ/WRITE
FNAMPN  EQU     $-VARS
        DS.L    1        ;LOAD FILENAME
LOADPN  EQU     $-VARS
        DS.L    1        ;LOAD ADDRESS
DIRPN   EQU     $-VARS
        DS.L    1        ;DIRECTORY ENTRY
RECCNT  EQU     $-VARS
        DS.L    1        ;LOAD RECORD COUNTER
RTNADD  EQU     $-VARS
        DS.L    1        ;RETRUN ADDRESS
DRIVE   EQU     $-VARS
        DS.L    1
SECTOR  EQU     $-VARS
        DS.L    1

        END
;
;    To write this to disk use the following commands
;        A>DEBUG          ;load debugger
;        -F100 1000 00    ;clear memory
;        -NBOOT.HEX       ;name of source file to load
;        -L               ;load file into memory at 100hex
;        -WCS:400 1 0 1   ;write memory starting at 400 hex
                          ;write drive B starting with logical
                          ;sector zero and writes 1 sector
;        -Q               ;exit to system
;
```

program can read and write data to any given sector. You can also modify files and save them by file name. To prepare the boot disk you will need to do both operations. The first step is to prepare the boot file. I used my favorite editor on the sample BIOS supplied by Hawthorne and pared it down to the essential items, and then used the calls to the PROM to load individual sectors.

There is another good reason to start with the bootloader first, it is simple, and it will show you the special considerations needed in the 68000 assembly language. Now I think the 68K is a lot easier to program than Intel chips, but the structure does require you to remember some simple principles. The 68K is a 32 bit machine and can address data either as 8, 16, or 32 bits. In assembly we use .B, .W and .L respectively for BYTE, WORD, and LONG. I got sloppy copying code from the Sage boot loader and shifted a .W for a .L. Depending on the operation this might give you the proper value, but then it might just give you all zeros instead. I kept sending those zeros until I realized what was going on.

It gets more complex when we talk about position independent code. In position independent code, all loads and stores are done off of values stored in address registers. These become base addresses and you offset or point to a memory location off of that register. Words point to the first 16 bits, with the values going to the lower 16 bits of the destination. The same operation as a Long will load the first 16 bits as high values, then the next 16 as low values. This problem became very important when calling Sage routines, as they are values pushed onto the stack. You can push ((A7)-) or pop ((A7)+) values as either words or longs, but whatever you do, both ends must be the same. I messed up and pushed some longs that should have been words, only to have unsuccessful reads.

Doing it

I have supplied some code showing what the boot loader is like, and the number of routines needed. Included with the sample is the dialog used with DEBUG to get the files on the disk. The steps go like this for the boot loader: save disk drive value (to make sure we continue to boot from it); print a message so we know we got this far; load the first DIR sector (remember the FAT was loaded with BOOT loader); find the file name and sector needed for loading the BIOS file; load those sectors; jump to the BIOS. You could save some time if you knew exactly which sectors to load, but then every time you made a minor change, the boot loader would need changing. Putting in messages may seem a luxury, but for systems that don't come up, knowing which routine failed become very important. A common way, and one possible here, is just outputting carriage returns and linefeeds. In the Sage that is a simple call or JSR to the PROM.

The Sage PROM deals with sector locations as blocks, and does not use track or side information. Their sector numbers start with ZERO and not ONE so you need to watch out for that. This made it simple as the record number, becomes the block number, which becomes the sector number and gets passed to the disk read routine. The Sage books talk about different formats, but I found that not to be true. I had forgot that the block numbers start at zero also, and had subtracted one from the record number (sectors start at 1,

Test Program

```
;
;        BIO LOADER TEST PROGRAM
;
;        USED TO SEE IF BOOTSTRAP LOADER WORKS
;        RENAME FILE TO SYSTEM.BIO FOR LOADING
;        AT 0B00H USES PROM TERMINAL I/O FOR
;        SAYING IT GOT THERE PROPERLY.....
;
BIOS_CODE        EQU      00000A00H         ;PROGRAM START
TERMTEXT         EQU      00FE0018H         ;TERMINAL STRING
TERMCRLF         EQU      00FE001CH         ;CRLF AT TERM
DEBUG            EQU      00FE0010H         ;DEBUG ENTRY
;
         ORG      BIOS_CODE
;
;
         JSR      TERMCRLF
         LEA      MSG1,A0
         JSR      TERMTEXT
         JSR      TERMCRLF
;
         JMP      DEBUG
;
MSG1     DC.B     "BIOS PROGRAM LOADED ",0
;
SCRATCH DC.L      1
;
         END
;
;     TO LOAD THIS PROGRAM USE MSDOS DEBUG AND THE FOLLOWING
;          A>DEBUG
;          -F100 3000 00     ;FILL MEMORY WITH ZEROS
;          -NBIO.HEX         ;NAME OF FILE TO LOAD
;          -L                ;LOAD FILE INTO MEMORY
;          -MA00 2000 CS:100 ;MOVE FILE STARTING AT A00 HEX
                             ;WHICH IS LOADING ADDRESS OF THE BIOS
                             ;PROGRAM, MOVING IT IN MEMORY TO 100 HEX
                             ;FOR PROPER SAVING TO DISK
;          -RCX              ;TELL SYSTEM HOW MUCH TO WRITE
;           CX: 200          ;SAVE ONE SECTOR TO DISK
;          -NSYSTEM.BIO      ;TELL NAME TO SAVE UNDER
;          -W                ;WRITE IT TO DISK
;          -Q                ;QUIT DEBUG
;     FOR THE FINAL BIOS USE RCX AND SET CS:??? TO LENGTH OF BIOS
;     TYPICALLY 1800 HEX LONG IF USING A00 TO 1FFF HEX.
;
```

records/blocks go from 0). I found that out after trying to load a simple BIOS test program (also included) and found it 200hex later in memory. This explains why any disk failures should return you to your debugger so you can check memory before a reset destroys what did happen.

The assembler I use was supplied by Hawthorne and assembles into Intel Hex format. The PC DEBUG will load those programs and you can move them around before letting it save them to disk. This assembler worked fine and only gave me problems once. I had incorrectly defined values in a table (used DS.L not DC.L) which changed the program counter which then caused all branch instructions to be out of range. That shows that it does check for programmer mistakes, which helps us rusty old dogs.

Closing

I am running a bit long, so I will try and tie up loose ends now. After the boot loader worked, I had the BIOS running (well sort of) in one day! I spent about a week studying the Sage code and K-OS samples, then a week programming the bootloader. I then took the boot loader and added terminal, printer, and a fuller disk I/O operation and used it as the BIOS. This was still making calls to the PROM and loading sectors one at a time (2 minutes to load the system), but it showed me it worked and that I was on the right track. Next I need to do the disk I/O in the BIOS with track and sector activity. I may later go back and change the BOOT loader to load the BIOS in one move, speeding that operation up. Later also I will put in interrupts and clock action, but then I will have the K-OS running and not be using the PC system.

A few fine points which need to be stressed are program locations. The boot loader in the Sage must go at 400hex. I Allocated buffer space, by putting the BIOS at A00hex. The BIOS must include or load a jump table 100hex lower than the OPERATE.BIN location. Until you have a chance to recompile the jump locations to routines into the BDOS or OPERATE.BIN, it will look for them 100hex below the starting location. Both the command and operate programs are position independent code so they can be anywhere, so could your BIOS. The only MUST do is put that jump table below OPERATE.BIN and COMMAND.BIN just above operate. I wrote and saved my BIOS as one file starting at A00hex and ending at 1FFFhex. That included the jump table and pre-zeroing out of variable memory locations (done by the assembler).

There are some other items that you must also learn about concerning the jump table. Each routine has certain items that must occur when the routine is jumped to. Typically items are pushed onto the stack (this case A4) and some status value returned on the stack after completion. Some routines must have this action, otherwise the system will go to never never land. I made lists and tables to help me out here as the manual is incomplete in this respect. I will go into more detail next time on these important steps. Till next time, read the manual and remember that the OPERATE.BIN is a HTPL program. HTPL programs must preserve registers A7 through A3 and D7. Your BIOS must not change these registers. Some variables and parameters are supplied by the BDOS as pointed to by A6. Read the HTPL user manual and pay close attention to the assembly language section.

This is by no means a complete coverage of everything needed to bring up the Sage or K-OS ONE. My major problem was choosing a direction to start with (I could have brought it up under the Sage's p-system), but once I started things fell into place easily. Next time I will cover more details about the BDOS and operating system. I will be contacting Joe about supplying more "how I did it" details as well as how he is doing on his installation manual. I am sure I missed something that you might not understand, so write us here at TCJ, and I will answer it next time. ■

The ZCPR3 Corner

by Jay Sage

In my last column I said that I would discuss the progress I have been making with a new version of ZEX, the memory-based batch processor for ZCPR3. As frequently happens, however, another subject has come up and preempted my attention. I thought I would still get to ZEX later in this column, but the column is already by far the longest I have written. So ZEX and other matters planned for this time will have to wait. For ZEX that is not a bad thing, since I still have a lot of work to do on it, and by two months from now there should be considerably more progress.

L'Affaire ARUNZ: J'Accuse

Not too long ago in a message on Z-Node Central, David McCord—sysop of the board and vice president of Echelon—leveled a serious accusation against me: that I have failed to provide proper documentation for my ARUNZ program. I immediately decided to mount a vigorous defense against this scurrilous charge using all the means at my disposal, including the awesome power of the press (i.e., this column in The Computer Journal).

Unfortunately, I find my defense hampered by a small technicality. True, many other people, faced with this very same impediment, have seemingly not been discouraged in the slightest from proceeding aggressively with their defense. However, I lack the character required to accomplish this. What is this technicality? It is the fact that the charge is true.

Excuses, Excuses

An effective defense being out of the question, perhaps I can at least offer some lame excuses.

First of all, it is not as true as it seems (if truth has degrees) that I have provided no documentation. There is a help file, ARUNZ.HLP, that at this very moment resides, as it has for years, in the HELP: directory on my Z-Node RAS (remote access system). Way back when ARUNZ was first made available to the user community, Bob Frazier was kind enough to prepare this help file for me, and it was included in the LBR file that I put up on my Z-Node. As a series of upgraded versions appeared, I began to omit the help file to avoid duplication and keep the new LBR files as small as possible. After a while, of course, the original library that did include the help file was removed from RASs. Hence the impression that there is no documentation. Of course, by now that help file is rather obsolete anyway.

If you are observant, you may have caught in the previous paragraph the deliberate circumlocution "made available to the user community." Why did I avoid the shorter and more natural expression "released?" Because ARUNZ has still to this day (more than two years—or is it three now—after its first 'availability'), not actually been released. Why? Because I still have not finished it. It is still in what I consider to be an incomplete, albeit quite usable, state. A few more tweaks, a couple of additional features, a little cleaning up of the source code, a detailed DOC file ... and it should be ready for a full, official release.

ARUNZ is, regrettably, not my only program that persists in this state. It is simply the oldest one. ZFILER and NZEX (new ZEX) suffer similarly. One might even say that this has become habitual with me. What happens, of course, is that I don't find the time to code that one little crucial additional feature before some other pressing issue diverts my attention. And by the time I do get back to it, I have thought of still another feature that just has to be included before the program should be released.

One solution would be to not make the programs available until they are really complete. There are two reasons why I have rejected this approach. First of all, though not complete to my satisfaction, the programs are in quite usable condition. It would be a shame if only I—and perhaps a small group of beta testers—had been able to take advantage of the power of ARUNZ during these two or three years.

The second problem with holding the programs back is that a lot of the development occurs as the result of suggestions from other users, who often have applications for the program that I never thought of and would never think of. In a sense, I have chosen to enlist the aid of the entire user community not only in the testing process but also in the program development process. And I think we have both benefited from this arrangement.

The procedure I have developed for keeping track of these 'released' test versions is to append a letter to the normal version number. As I compose this column, ARUNZ stands at version 0.9G, ZFILER stands at 1.0H, and NZEX stands at 1.0D. When final versions are released, I will drop the letter suffixes (except for NZEX, which will become ZEX version 4.0).

The usability of the programs is probably the fundamental factor that keeps them in their incomplete state. When one of them has some serious deficiency or or simply begs for an exciting new feature, it gets my attention. Once it is working reasonably well, however, I can ignore it and devote my attention to other things that badly need fixing. That is how I recently got started on NZEX.

Making Amends

Since excuses, no matter how excusing, do not solve a problem, I will take advantage of this column to make amends for the poor state of the ARUNZ documentation by providing that documentation right here and now. I hope it will lead more people to make more complete and effective use of ARUNZ, which for me has been the single most powerful utility program on my computers.

To understand ARUNZ, one must first understand the concept of the ZCPR alias, and to understand aliases one must understand the multiple command line facility. I have written some things about these subjects in earlier columns, notably in issues #27 and #28, but I will start more or less from the beginning here.

Multiple Command Lines

One of the most powerful features of ZCPR3 is its ability to accept more than one command at a time and to process these commands sequentially. Quoting from my column in TCJ issue #27: *The multiple command capability of Z System ... is important not so much because it allows the user to enter a whole sequence of commands manually but rather because it allows other programs to do so automatically.*

Obviously, in order to process multiple commands, the list of commands (at least the unexecuted ones) must be stored in some secure place while earlier ones are being carried out. In the case of ZCPR3, there is a dedicated area, called the multiple command line (MCL) buffer, located in the operating system part of memory. It stores the command line together with a pointer (a memory address) to the next command to be executed. Every time the ZCPR3 command processor returns to power, it uses the value of the pointer to determine where to resume processing the command line. Only when the end of the command line is reached does the command processor seek new command line input.

Storing multiple commands in memory is not the only possibility. Another secure place to keep them is in a disk file. This is in some ways what the SUBMIT facility does using the file $$$.SUB. The main drawback to this approach is the speed penalty associated with the disk accesses required to write and read this file. There is also always the possibility of running out of room on the disk or of the diskette with the $$$.SUB file being removed from the drive. Using a memory buffer is faster and more reliable.

Digital Research's most advanced version of CP/M, called CP/M-Plus, also provides for multiple command line entry, but it does it in a rather different, and I think less powerful, way. When a multiple command line is entered by the user, the system builds what is called a resident system extension (RSX), a special block of code that extends the operating system below its normal lower limit. This RSX holds any pending commands. But since it is not always present and is not at a fixed, known location in memory, there is no straightforward way for programs to manipulate multiple command lines. On the other hand, this method does provide a bigger TPA when only single commands are entered.

In a ZCPR3 system, the MCL has a fixed size and is in a fixed location. Moreover, a ZCPR3 program can find out where the MCL is located by looking up the information about it in the ZCPR3 environment descriptor (ENV), another block of operating-system memory containing a rather complete index to the features of the particular ZCPR3 system. The location of the ENV is the one key fact that is conveyed to all ZCPR3 programs. Prior to ZCPR version 3.3, the address of the ENV had to be installed into each program manually by the user before the program could be used; with ZCPR33 this installation is performed automatically by the command processor as the program is run.

The Alias Program

One of Richard Conn's brilliant concepts in designing ZCPR3 was the utility program he called ALIAS, whose function is to create COM files that, in turn, build multiple command lines and insert them into the MCL buffer. When ALIAS is run, it prompts the user for (1) the name of the alias file to create and (2) a prototype command line, nowadays called a script. When the resulting COM file is run, it takes the script, uses the information in it to construct a complete command line, and then places that command line into the MCL buffer so that the commands it contains will be run.

The simplest script would be nothing more than a completely formed command line. For example, if we wanted to have a command (COM file) that would display the amount of free space on each of drives A, B, and C, we could make an alias SPACE.COM containing the script

```
SP A:;SP B:;SP C:
```

We assume here that our RCP (resident command package) includes the SP (space) command.

Such a script can have only a single purpose. Much more powerful capability is provided when the script can contain parameter expressions that are filled in at the time the command is run. The aliases produced by ALIAS.COM support a number of parameter expressions, including the $1, $2, ... $9 parameters familiar from the SUBMIT facility. An alias called ASMLINK with a script containing the following command sequence

```
SLR180 $1
IF ~ER
SLRNK /A:100,$1/N,$1,VLIB/S,Z3LIB/S,SYSLIB/S,/E
FI
```

can then be used to assemble and (if there were no errors in assembly) link any program. The expression $1 is replaced by the first token on the invoking command line after the name of the alias. A token, we should note, is a contiguous string of characters delimited (separated) by a space or tab character. Thus with the command

```
ASMLINK MYPROG
```

the string "MYPROG" will be substituted for each of the three occurrences of the expression "$1" in the script to form the command line. Any commands in the MCL after the alias command are appended to the expanded script.

The Advent of ARUNZ

One day it suddenly struck me that Conn-style aliases are extremely inefficient with disk space. Each one contains, of course, the prototype command line (the script), which is unique and essential to each alias, but which is at most about 200 characters long and often much less (17 and 67 in the two examples above, if I counted right). But each one also contains a complete copy of the script interpreter and command line manipulation code, about 1K bytes long, which is exactly the same in each alias. Why not, I thought, separate these two functions, putting all the scripts into a single, ordinary text file (ALIAS.CMD) and the alias processing code in another, separate file (ARUNZ for Alias-RUN-Zcpr)?

Because there is only a single copy of the ARUNZ code in the system rather than a copy of it with each alias, I felt that I could afford to expand the code to include many additional features, in particular much more extensive parameter expansion capability. These features will be described later.

The ALIAS.CMD File

Let's begin by looking at the structure of the ALIAS.CMD file. First, we should make it clear that ALIAS.CMD is a plain, ordinary text file that you create using your favorite text editor or word processor (in non-document mode).

Each physical line in the file contains a separate alias definition. At present there is no provision for allowing definitions to run over to additional lines, so for long scripts your editor has to be able to handle documents with a right margin of more than 200 characters. As I sit here composing this column, it occurs to me that a nice solution to this problem might be to allow the ALIAS.CMD file to be created by a word processor in document mode and to have WordStar-style soft carriage returns be interpreted by ARUNZ.COM as line-continuation characters. I will experiment with that possibility after I finish this column, and if it works there may be an ARUNZ version 0.9H by the time you are reading this.

Each alias definition line contains two parts. The first part, the name field, defines the name or names by which the alias will be recognized, and the second part, the script field, contains the script associated with that name or those names.

The name field must start in the very leftmost column (no leading spaces), and the two fields are separated by a space or tab character. Thus ALIAS.CMD might have the following general appearance:

```
FIRST-NAME-FIELD    first script
NEXT-NAME-FIELD     next script
...                 ...
LAST-NAME-FIELD     last script
```

For ease of reading, I follow the convention of putting the alias name field in upper case and the script strings in lower case, but you can use any convention (or no convention) you like, since ARUNZ does not generally care about case (the sole exception will be described later).

To make the ALIAS.CMD file easier to read, you can include comment and formatting lines. Blank lines are ignored and can be used to separate groups of related alias definitions. Also, any line that begins with a space (no name field) will never match an alias name and will thus have the effect of a comment line. You can use this to put titles in front of groups of definitions.

To tell the truth, I always wanted to be able to format the ALIAS.CMD file as I just described, but I never got around to adding the code to allow it. As I was sitting here writing just now, I suddenly decided to see what would happen if the ALIAS.CMD file contained such lines. With BGii in operation, a quick " from the keyboard took me to the alternate task, and I gave it a whirl. Imagine my surprise and delight to discover that the formatting already works! No new code is required.

The Name Field in ALIAS.CMD

The name field can contain a simple name, like SPACE or ASMLINK, but more complex and flexible forms are also supported. First of all, the name field can consist of any number of individual name elements connected by an equal sign (with no intervening spaces, since a space would mark the end of the name field). Thus a line in ALIAS.CMD might have the following appearance:

```
NAME1=NAME2=NAME3          script string
```

Secondly, each name element can represent multiple names. There are three characters that have special meanings in a name element. The first is a question mark ('?'). As with CP/M file names, a question mark matches any character, including a blank space. Thus the alias name DIR? will match any of the following commands: DIR, DIRS, DIRR, and so on.

The second special character is currently the period ('.'). For reasons that I will not go into here (having to do with a new feature under consideration for ZCPR34), I may change this to another character (perhaps the asterisk), so check the update documentation with any version of ARUNZ beyond 0.9G. The period does not match any character, but it signals the comparison routine in ARUNZ that any characters after the period are optional. If characters are present in the command name, they must match those in the alias name, but the characters do not have to be present. For example, the alias name field

```
FIND.FILE=FILE.FIND
```

will match any of the following commands (and quite a few others as well): FIND, FINDF, FINDFILE, FILE, FILEF, FILEFIND. It will not, however, match FILES or FINDSTR or FINDFILES.

I have never had any occasion to make use of the capability, but the two special characters can be combined in a single name element. Thus FIND.FI?E matches FINDFILE and FINDFIRE but not FINDSTR, and ?DIR.R matches SDIR, SDIRR, XDIR, and XDIRR (but not DIR). I think you can see that the special characters allow for very compact expressions covering many names.

The third special character is the colon (':'). If any name element begins with a colon, then it will match any alias name whatsoever. This is called the default alias, the alias to be run if no other match is found. Since ARUNZ scans through the ALIAS.CMD file from top to bottom searching for a matching name, if the default name is used at all, it makes sense only as the last alias in the file, since no alias definitions in lines below it can ever be invoked. Note that letters after the colon have no significance; you may include them if you wish as a kind of comment.

One possible use for the default alias would be a line like the following at the end of the ALIAS.CMD file:

```
:DEFAULT echo alias $0 not found in alias.cmd
```

If no specific matching alias is found, this default alias will report that fact to the user as a kind of error message. I do not recommend using the default alias in this way, however, because it will interfere with ZCPR33's normal invocation of the error handler when ARUNZ has been set up as the extended command processor (ECP) and a bad command is entered.

There is one use of the default alias that can augment the extended command processing power of ZCPR33. When ARUNZ has been set up as the ECP and a command is found neither as a system command, nor COM file, nor ARUNZ alias, one might want to try running the command from COMMAND.LBR using the LX program. This is a kind of chained ECP operation. ARUNZ is the first ECP; LX is the second. This can be accomplished, using version 1.6 or later of LX, by adding the following line at the end of the ALIAS.CMD file:

```
:ECP-CHAIN lx / $0 $*
```

The meaning of the parameters $0 and $* will be explained later. With this default alias, if a command cannot be resolved by a specific ARUNZ alias, then an LX command line will be generated to search for a COM file with the name of the command in COMMAND.LBR. The special parameter '/' as the first command line parameter to LX tells LX, when it cannot resolve the command either, to pass to the ZCPR3 error handler only the user command line (i.e., to omit the "LX / " part of the command).

This might be a good time to note that ARUNZ alias names are not limited to only eight characters or to the characters allowed in disk file names. For example, you have a perfect right to define an alias with the name FINDFILES (nine letters) and to invoke it with the command ARUNZ FINDFILES. If ARUNZ has been set up as your extended command processor (see my book *The ZCPR33 User Guide* for a discussion of ECPs), then when you enter the command FINDFILES, the command processor will first look for a disk file FINDFILE.COM, since it truncates the command name to eight characters. If this file is not found, the command processor will then, in effect, run ARUNZ FINDFILES, including all nine characters. I have not thought of any uses for aliases with control characters in their names, but you can define such aliases if you wish.

Another fine point to be noted is that both leading blank spaces and an initial colon are stripped from the command name before scanning for a matching alias name. It is obvious that if leading blanks were not stripped, a leading blank would prevent any match from being found. The colon is stripped so that a command entered as ":VERB" will match an alias name of "VERB" without the colon. If a directory specification is included before the colon, it will not be stripped. When the BADDUECP option is enabled in the configuration of ZCPR33, this allows illegal directory specifications to be passed to ARUNZ for processing.

The Script Field in ALIAS.CMD

The script field in the ALIAS.CMD file contains the prototype command line to be generated in response to a matching alias name. The script contains three kinds of items:

(1) characters that are to be put into the command line directly.

(2) parameter expressions that ARUNZ is to evaluate and convert to characters in the command line.

(3) directives to ARUNZ to perform special operations.

There is nothing that has to be said about the first class of characters. They comprise any characters not covered by the other two sets. The simple example of the SPACE alias, which would appear in ALIAS.CMD as:

```
SPACE sp a:;sp b:;sp c:
```

has only direct characters. There are no special directives and no parameters to evaluate.

ARUNZ Parameters

ARUNZ supports a very rich set of parameter expressions, which we will now describe. As rich as the set is, there are still important parameters that need to be added. Some of these will be mentioned later in the discussion. First let's see what we can already do.

Parameters begin with either a caret ('^') or a dollar sign ('$'). The former is quite simple; it is used to signal a control character. The ASCII representation of the character following the caret is logically ANDed with 1FH, and the result is placed into the command line. Of course, control characters other than carriage return and line feed can equally well be placed directly into the script.

At present there is no trap to prevent generating a null character (caret-space will do this: space is 20H, and 20H & 1FH = 00H). If this is used, the resulting null will effectively terminate the command line. Any characters that come after the null character will be ignored by the command processor. This could conceivably be useful for deliberately cancelling pending commands in a command line, but I have never used it. In fact, I was surprised to find that I did not have a trap for it. On thinking about it now, however, it seems best to continue to allow it. Just "user beware!" when it comes to employing it.

Parameters introduced by a dollar sign provide much more varied, interesting, and powerful capabilities. The special ARUNZ directives are also introduced by a dollar sign. A complete list of the characters that can follow the dollar sign, grouped by function, is given below. Detailed discussion of each will follow.

```
$   ^
*   -
digit  (0..9)
D  U  :  .
F  N  T
'  '
R  M
I  Z
```

Character Parameters

The parameters '$' and '^' are provided to allow the two parameter lead-in characters to be entered into the command line text. Many users, present company unhappily included, have made the mistake of trying to enter a dollar sign directly into the alias script. If this is done, the dollar sign is (mis)interpreted as a parameter lead-in character. You must put '$$' in the script to get a single dollar sign in the command line.

The worst example I have seen (and committed) of this kind of error is in a command like "PATH A0 $$ A0". This looks perfectly reasonable and does not produce any kind of error message when it runs (as "PATH A0 $0 A0" would, for example, when $0 got expanded to 'PATH'). Unfortunately, it runs as "PATH A0 $ A0", where the single dollar sign now means current-drive/user-0 (this is perhaps a flaw in the way the PATH works, but that is the way it is). The proper form of the script is:

```
PATH A0 $$$$ A0
```

where each pair of dollar signs turns into a single dollar sign.

Complete Command-Tail Parameters

The parameters '*' and '-' refer to entire sections of the command line tail. The asterisk represents the entire tail exactly as the user entered it. The parameter expression $-n, where 'n' is a number from 0 to 9, represents the command tail less the first 'n' tokens (a token was defined earlier). The parameter $-0 has the same meaning as $*.

Many users have confused 'command line tail' with 'command line'. The two are not the same. A command line consists of the command name (the 'verb') and the tail. Thus the command line tail is the command line less the first token. Perhaps some examples will help. Suppose the command line is

```
command token1 token2 token3 token4
```

Then

```
$*  = "token1 token2 token3 token4"
$-2 = "token3 token4"
$-4 = ""
```

Note that $-4 is the null string; that is, $-4 will be replaced by no characters at all.

Also note that there is no leading space in the string assigned to $*. ALIAS.COM (and the earliest version of ARUNZ, I believe) had a bug in this respect in that it did include the leading space in the command line tail, since that is how the tail is stored by the command processor in the buffer beginning at memory address 0080H. The script "find $*" when invoked with the tail "string" then became "find string" with two spaces between "find" and "string". In such a case, Irv Hoff's FIND program failed to work as expected, probably because it was looking for " string" with a leading space.

Complete Token Parameters

The digit parameters '0' through '9' represent the corresponding token in the command line that is being parsed. In the example command line above the digit parameters have the following values:

```
$0 = "command"
$1 = "token 1"
$5 = ""
```

Except for the '0' parameter, these parameters are familiar from the CP/M SUBMIT facility. The expression $0 is an extension used to represent the command verb itself. Just think of the tokens on the command line as being numbered in the usual computer fashion starting with zero instead of one. A token that is absent from the command line returns a null string (no characters) as with $5 in the above example.

As just mentioned, many users confuse the command line tail and the command line. If you want only the tail, use the parameter $*. If you want to represent the entire command line, use the expression "$0 $*". Most often it is the command line tail that is to be passed to a command, and the ALIAS.CMD line will read something like

ALIAS realverb $*

This is a direct implementation of the common meaning of 'alias' as another name for something. When ALIAS is invoked, we simply want to substitute 'realverb' for it while leaving the command tail as it was.

There are other occasions, however, as with the LX default alias example given earlier, where the entire command line must be passed. There are still other occasions, such as in the first default alias example above, where only the name of the verb used is needed. Because a given script in ALIAS.CMD can correspond to many possible alias names, it is important to have a parameter that will return the name that was actually used in any particular instance.

Token Parsing Parameters

There are many instances in which it is extremely useful to be able to break any token down into its constituents. The parameters 'D', 'U', ':', and '.' do this. They assume that the token is in the form of a file specification, which may have (1) a directory specification using either a named directory or a drive and/or user number; and/or (2) a file name; and/or (3) a file type. Each of the four parameters above is followed by a number from 1 to 9 to designate the token to parse ('D' and 'U' can also have a 0). After discussing each one individually, we will give some examples.

The parameter 'D' returns the drive specified or implied in the designated token. If there is no directory specification or if only a user number is given, then $Dn returns the default (logged) drive at the time ARUNZ constructs the command line.

WARNING—NOTE WELL: this is not necessarily the drive that will be logged in at the time when that part of the command actually executes!! This, too, has been the source of grief in the use of ARUNZ. ARUNZ has no infallible way to know what directory will be logged in when some future command runs; it only knows what directory is the default directory at the time ARUNZ itself is running.

The 'U' parameter is similar in all respects to 'D', and the same warning applies. The parameters $D0 and $U0 can also be used. They always return the default drive and user at the time ARUNZ interprets the script.

The parameter ':' represents the file name part of the token, while the parameter '.' represents the file type part of the token. One way to remember the characters for these two parameters is to think that colon stands for the part of the token after a colon and period stands for the part of the token after a period. Admittedly, 'N' for name and 'T' for type would have been more sensible, but as we shall see shortly, these are already used for something else.

Generally speaking, the entire token can be represented as

DnUn:$:n.$.n

where 'n' is a digit.

Let us consider some examples. Suppose the following command is entered at the prompt:

```
B1:WORK>command root:fn1.ft1 c:fn2 2:.ft3
```

and that COMMAND.COM is not found, so that the command is passed on to ARUNZ and the extended command processor. Also assume that the ROOT directory is A15. Then here are the values of the parameters for the four tokens in the command:

```
$D0 = "B"        $U0 = "1"        $0  = "COMMAND"

$1  = "ROOT:FN1.FT1"
$D1 = "A"        $U1 = "15"       $:1 = "FN1"      $.1 = "FT1"

$2  = "C:FN2"
$D2 = "C"        $U2 = "1"        $:2 = "FN2"      $.2 = ""

$3  = "2:.FT3"
$D3 = "B"        $U3 = "2"        $:3 = ""         $.3 = "FT3"
```

Note the value of the following parametric expression:

$D1$U1:$:1.$.1 = "A15:FN1.FN2"

You can see that the 'D' and 'U' parameters can be used to convert a named directory into its drive/user form.

System File Name Parameters

The ZCPR3 ENV contains four system file names, each with a name and a type. These file names, numbered 0..3, are used by various programs, especially shells. VFILER and ZFILER, for example, keep the name of the file currently pointed to in system file name 1. These file names can also be read and set using the utility program SETFILE.

The parameters 'F', 'N', and 'T' followed by a digit from 0 to 3 return, respectively, the entire filename (name.typ), file name, and file type of the specified system file.

User Input Parameters

The single and double quote parameters are used for prompted user input. The forms of the parameter expressions are:

$"prompt" or $'prompt'

When the parameter $" or $' has been detected, any characters in the script up to the matching parameter character or the end of the script line are echoed as a prompt to the user's screen. These characters are echoed exactly as they appear in the script; no conversion to upper case is performed. The prompt string for the double quote parameter can contain single-quote characters, and the prompt string for the single quote parameter can contain double-quote characters. There is, at present, no way to include the type of quote character used as the parameter in the prompt string.

After the prompt has been output to the console, ARUNZ reads in a line of input from the console (user input). At this point there is a subtle but important distinction between the two user input parameters. The single quote form takes the entire text string entered from the console and places it in the command line. In particular, this input may contain semicolons, allowing the user to enter multiple commands. The double quote form ignores a semicolon and any text thereafter. This is intended for secure systems, where it prevents the user, when prompted for a program option, from slipping in complete additional commands.

One pitfall to which many users have succumbed is the failure to appreciate that the user input parameters perform their function at the time that ARUNZ is running and interpreting the script, not when the program in the command line is running. Consider the alias definition:

```
ERAFILE dir $1;era $"File name to erase: "
```

The intention here is to first display a list of the files that match the first command line token and then to allow the user to enter the one to be erased. This is not what will happen. ARUNZ will put up the prompt "File name to erase: " at the time the command line is being built, i.e., before DIR is run. The prompt will come before the directory display.

The way around this problem is to use two ARUNZ aliases as follows:

```
ERAFILE      dir $1;/eraprompt
ERAPROMPT       era $"File name to erase: "
```

Now when ERAFILE is run, it will display the directory and then run the command "/ERAPROMPT". The slash here is a ZCPR33 feature that indicates that the command should be sent directly to the extended command processor. This saves the time that would otherwise be wasted searching for a file named ERAPROMPT.COM (actually, ERAPROMP.COM, since the ninth character will be truncated from the name). If you are not running ZCPR33 (but you should be!!) or are running BGii, use a space instead. This will work with both ZCPR33 and BGii and will have no effect in ZCPR30. I am using the slash in the examples because a space is hard to see in print. When ERAPROMPT runs and the user is prompted for the name, the directory listing will already be on the screen.

Whenever console input is requested by any program, one must keep in mind the possibility that ZEX will be running and consider the question of whether the input request should be satisfied from the ZEX script or by direct user input. ARUNZ is configured, in the absence of a specific directive to the contrary, to turn ZEX input redirection off during ARUNZ prompted input. Thus, even if ZEX is running at the time ARUNZ is invoked, the user input parameters will request live user input.

If you do want ZEX to be able to provide the response to ARUNZ prompted input automatically from the ZEX script, then you must include the ARUNZ directive $I ('I' for input redirection) before the $" or $' parameter. The $I directive is effective only for the next user input operation. After each prompted user input operation, the default for ZEX input redirection is turned off. The $I directive need not immediately precede the $" or $' but there must be a separate $I for each input requested.

Register and Memory Parameters

Two parameters are provided for referencing values of the ZCPR3 user registers and the contents of any memory location in the system.

By Richard Conn's original specification, there were ten user registers numbered from 0 to 9. However, the block of memory in which those ten registers fall is actually 32 bytes long. Conn designated the last 16 bytes of this block as 'user definable registers', but he and others later used them in programs such as Term3 and Z-Msg. As a result, one has to be very careful in making use of them. The last 6 bytes of the first half of the block were defined as 'reserved bytes'. Various uses have been made of them as well.

The ARUNZ parameter 'R' can reference any of the first 16 bytes using the form $Rn, where 'n' is a hexadecimal digit. The decimal digits reference the true user registers, and the additional digits 'A' through 'F' reference the reserved bytes. In the current version of ARUNZ, the value is returned as a two character hexadecimal value. However, I would like to provide in the future a way to return the value in either decimal or hexadecimal form. A complication with the decimal form is the need to indicate the format: one character, two characters with leading zeros, three characters with leading zeros, or the number of characters required for the particular value with no leading zeros.

One of the uses I envisioned for this parameter, though I have never actually used it this way, is for automatic sequential numbering of files. Thus a script might include the string "copy $:1$r3.$.1 = $1;reg p3". This would copy the working file given by token 1 to a new file with the hex value of register 3 appended to the file name. For a file name of PROG.Z80 this might be PROG03.Z80. Then the value of register 3 would be incremented so that the next file name in sequence (PROG04.Z80) would be used the next time the alias was invoked.

The parameter 'M' is used in the form $Mnnnn, where 'nnnn' is a precisely four-digit hexadecimal address value. The parameter returns the two character hexadecimal value of the byte at the specified memory address. I use this on my RAS to determine if the system is running in local mode. The BDOS page at address 0007H has a different value when BYE is running. There might be a script of the form:

```
if eq $m0007 c6;....;else;echo not allowed in remote mode;fi
```

The commands represented by the ellipsis "...." will run only if in local mode (BDOS apparently located at page C6H).

ARUNZ Directives

There are presently two ARUNZ directives. We have already discussed one of them, 'I', under the user input parameters. The other one is 'Z'.

Ordinarily, once ARUNZ has interpreted the alias script and evaluated the parameters, it appends to the resulting command line any commands in the multiple command line buffer that have not already been executed. This is usually what one wants. There is one possible exception.

As I discussed in issues #27 and #28 of *The Computer Journal*, one sometimes wants an alias to invoke itself or other aliases recursively. This can sometimes lead to problems with the build up of unwanted pending commands that eventually causes the command line to overflow the buffer space allowed for it. In such a case one might want only the current expanded script command line to be placed in the MCL, with any pending commands dropped. A $Z directive anywhere in the script will cause ARUNZ to do this. Note that the directive is not a toggle; multiple uses has the same effect as a singe use. Remember, however, that Dreas Nielsen's alias recursion technique, described in issue #28 and in examples below, is generally preferable to the technique using $Z.

Applications for ARUNZ Aliases

In this section I will use a number of sample scripts to illustrate various ways in which one can make use of the power of ARUNZ aliases. I'm sure there are many I have not thought of, and I invite you to send me your suggestions and examples. In all cases I will be assuming that ARUNZ is the extended command processor (typically renamed to CMDRUN.COM).

In general, one can identify the following classes of alias applications:

(1) providing synonyms for commands.
(2) trapping and/or correcting command errors.
(3) automating complex operations into single commands.

Within the last category fall two special subclasses:

(a) performing 'get, poke, & go' operations.
(b) providing special functions like recursion and repetition.

Command Synonyms

The most basic use of aliases is to provide alternative names for commands. Here are some examples from my personal ALIAS.CMD file.

For displaying the directory of a library file, I now use the program LLF. However, after years of using LDIR, both before LLF was released and still on most remote access systems, I prefer to use that name and have renamed LLF.COM to LDIR.COM. Sometimes, however, I forget or want to be sure I am running LLF and enter the command LLF explicitly. Then I am saved by the alias line:

```
LLF ldir $*
```

Similarly, I have recently begun to use LBREXT instead of LGET. LGET is easier to type, and I am used to it, so I have the alias:

```
LGET lbrext $*
```

LBREXT is so new that I did not want to rename it to LGET, since I might too easily forget which program the disk file really is. I know I never have the old LDIR.COM around any more. In both of these examples, the alias simply substitutes a different verb in the command line; the tail is left unchanged.

Before the advent of ZCPR33, when path searching always included the current directory, I would speed up the disk searching in these cases by including an explicit directory reference with the script. Thus the two commands above might be:

```
LLF  a0:ldir $*
LGET  a0:lbrext $*
```

This way the command processor would go straight to A0 no matter where I was logged in at the time.

With ZCPR33 one can bypass the path search for commands that one knows are in ALIAS.CMD by entering the command with a leading space or slash (assuming the usual configuration of ZCPR33). Sometimes I might try to outfox the system and, thinking LBREXT is the alias name, enter the command as '/LBREXT . . . '. So that this will work, I extend the alias lines to:

```
LLF = LDIR    a0:ldir $*
LGET = LBREXT  a0:lbrext $*
```

The command is an alias for itself!! Odd, but useful. It is a good idea if you do this, however, to be absolutely sure to include an explicit directory prefix before the command name in the script. If you don't, the following situation can arise. Suppose the alias line reads:

```
TEST test $*
```

but for some reason TEST.COM is not on the disk (or at least not on the search path). Now you enter the command TEST. The command cannot be found as a COM file, so the command processor sends it to ARUNZ. ARUNZ proceeds to regenerate the same command, which again cannot be found, and so on until you press the little red button or pull the plug. Not always to complete catastrophe, but definitely a nuisance. With ZCPR33, if the command has an explicit directory prefix, control is passed directly to the error handler if the COM file cannot be found in the specified directory. It figures that if you go to the trouble of specifying the directory, you must mean to look there only.

Another use for synonyms is to allow a short-form entry of commands. Here are two examples:

```
SLR.180  asm:slr180 $*
ED.IT    sys:edit $*
```

Synonyms are especially helpful on a remote access system or on any system that will be used by people who are not familiar with it or expert in its use. Consider, for example, the task of finding out if a certain file is somewhere on the system and where. Some systems use FINDF, the original ZCPR3 program for this purpose; others use one of the standard CP/M programs (WIS or WHEREIS); and others have begun to use the new, enhanced ZSIG program called FF. This can be very confusing to new users or to users who call many different systems. The solution is to provide aliases for all the alternatives. Suppose FF is the real program in use. Then the following line in ALIAS.CMD will allow all the forms to be used equally:

```
FINDF = WIS = WHEREIS ff $*
```

In fact, while I am at it, I usually throw in a few other forms that someone might try and that are sufficiently unambiguous that one can guess with some confidence that this is the function the user intended:

```
FIND.FILE = FILE.FIND = WIS = WHERE.IS = FF    a0:ff    $*
```

Note that this single alias, which occupies only 46 bytes in ALIAS.CMD (including the CRLF at the end of the line), responds to 8 commonly used commands for finding files on a system. Thus the cost is a mere 6 bytes per command!!

Trapping and Correcting Command Errors

Aliases can be used to trap commands that would be errors and either convert them into equivalent valid commands or provide some warning message to the user.

It is generally not desirable to have a very long search path, because every time a command is entered erroneously, the entire path has to be searched before the extended command processor will be brought into play. On my SB180 with its RAM disk, I sometimes want the path to include only M0:, the RAM disk directory. The RAM disk, of course, cannot contain all of the COM files I use. For COM files that reside on the floppy disk, I can include an alias.

For example, MEX.COM and all its associated files take up a lot of disk space, and I keep them in a directory called MEX on my floppy drive B. The ALIAS.CMD file can have the line:

```
MEX mex:mex $*
```

Without this alias I would have to remember to enter MEX:MEX manually. If I forgot, I would get the error handler and then have to edit the line to include the MEX: prefix. The 16-byte entry above in the ALIAS.CMD file saves me all this trouble.

Every computer user probably has some commands whose names he habitually mistypes (switching 'g' and 'q' for example or reversing two letters). My fingers seem to prefer 'CRUNHC' to 'CRUNCH', so I have the following alias line:

```
CR.UNHC crunch $*
```

Note that while I am at it, I allow the shorter form CR as well. My fingers like that even better.

On a remote access system there are many situations where correcting common mistakes can be handy. Richard Jacobson (Mr. Lillipute, sysop of the RAS that now serves TCJ subscribers) calls my Z-Node quite often. Either he has a Wyse keyboard with very bad bounce (as he claims) or he is a lousy typist (and refuses to admit it). When he wants to display a directory, his command is more likely to come out DDIR or DIRR than it is to come out correctly as DIR. So I added those two forms to my existing alias which allowed XD and XDIR (and /DIR); it now reads:

```
XD.IR = DDIR = DIR.R a0:dir $*
```

Compensating for Richard's keyboard stutter takes up only seven extra bytes on my disk, not a very big sacrifice to make for a friend!

Another example, one that is more than just a synonym for a mistyped command, is an alias that comes into play when a command becomes unavailable, perhaps because of a change in security level. The RCP may, for example, have an ERA command that is only available when the wheel byte is set. When the wheel byte is off, ZCPR33 will ignore the command in the RCP and forward an ERA command to the extended command processor or error handler (assuming there is no ERA.COM). You might want to trap the error before the error handler gets it using an alias such as:

```
ERA echo e% > rasing of files not allowed
```

When the wheel byte is set, the ERA command will execute normally (unless entered with a leading space or slash). When the wheel byte is off, the user will get the message "Erasing of files not allowed", which, unlike the invocation of an error handler, makes the situation perfectly clear.

It is obviously very hard for users to remember the DU forms for directories on a remote system, and that is one reason why named directories are provided. But even names are not always easy to remember precisely. Aliases can help by providing alternative names for logging into directories, provided ZCPR33 has been assembled with the BADDUECP option enabled so that invalid directory-change references are passed on to the extended command processor.

Suppose you have a directory called Z3SHELLS. One might easily forget the exact name and think that it is Z3SHELL or SHELLS or SHELL. The following line in ALIAS.CMD

```
Z3SHELL: = SHELL: = SHELLS: z3shells:
```

would take care of all of these possibilities. Note, however, that it will not help a reference like "DIR SHELL:". [If you wanted this to be accepted, you would have to go to considerable trouble. You might be able to go into the NDR (named directory register) and tack onto the end an entry for a directory named SHELL associated with the same drive and user as Z3SHELLS. All existing NDR editors will not allow a DU area to have more than one name, so you would have to use a debugger or patcher. If anyone tries this, let me know if it works.]

I occasionally slip up and omit the colon on the end of a directory change command (and users on my Z-Node do it surprisingly often). It is very easy for ARUNZ to pick this up as well and add the colon for you. Just include the following alias line:

```
Z3SHELL = Z3SHELLS = SHELL = SHELLS z3shells:
```

All of these aliases can be combined into the single script:

```
Z3SHELL.: = Z3SHELLS = SHELL.: = SHELL.S: z3shells:
```

Seven forms are covered by an entry of only 47 bytes, a cost of less than 7 bytes each. Note that the name element Z3SHELLS, unlike the other three name elements, does not allow an optional colon. If it were included and for some reason there were no directory with the name Z3SHELLS, you could get into an infinite loop.

On my Z-Node I provide a complete set of aliases for all possible directories so that any legal directory can be entered with or without colons and using either the DIR or the DU form. Thus, if Z3SHELLS is B4, the script above would be:

```
Z3SHELL.: = Z3SHELLS = SHELL.: = SHELL.S: = B4.: z3shells:
```

Before ZCPR33 came along and provided this service itself, I would allow callers to use the DU form to log into unpassworded named directories beyond the max-drive/max-user limits by including aliases of the above form. If the maximum user area were 3 in the above example, the commands "B4:" and "B4" would still have worked (even under ZCPR30) because ARUNZ mapped them into a DIR form of reference. Although this is no longer necessary with ZCPR33, a complete alias line like the one above covers all bases. The user can even enter any of the commands with a leading space or slash and they will still work.

Finally, I provide on the Z-Node a catch-all directory change alias to pick up directory change commands that don't even come close to something legal. At the end of ALIAS.CMD (i.e., after all the other directory-change aliases described above, so that they get the first shot at matching), I include the line:

```
?: = ??: = ???: = ????: = ?????: = ??????: = ???????: = ????????:     echo
d%>irectory  %<$0%>  is  not  an  allowed  directory.
%<t%>he ` m ^ j valid directories are:;pwd
```

Thus when the user enters the command "BADDIR:", he get the PWD display of the system's allowed directories prefixed by the message:

```
Directory BADDIR: is not an allowed directory. The valid directories are:
```

[Note the use of Z33RCP's advanced ECHO command with case shifting ('%< to switch to upper case and '%>' to switch to lower case) and control character inclusion (caret followed by the character).]

Automating Complexity

Complexity is a relative term, and in my old age (also relative) I enjoy the luxury of letting my computer perform as much labor on my behalf as it possibly can. We already saw how ARUNZ aliases can provide short forms for commands (CR for CRUNCH). It can also allow one to completely omit commands.

At work I have been maintaining a phone directory in a file called PHONE.DIR. I got tired of invoking my PMATE text editor using the command "EDIT A0:PHONE.DIR", so I added the following line to ALIAS.CMD:

```
PHONE edit a0:phone.dir
```

Now I just type PHONE and, bingo, I'm in the editor ready to add a new name. Similarly, I used to look up numbers for people using JETFIND as follows:

```
JF -gi smithones a0:phone.dir
```

This would give me, from any directory, a paginated listing of lines in PHONE.DIR containing either "smith" or "jones" (ignoring case). My poor tired fingers ache just thinking about all that typing. Now I have the alias line:

```
# = CALL = NUM.BER jf -gi $1 a0:phone.dir
```

Now a simple " # smith" puts Smith's number up on my clean CRT screen in a jiffy.

Here is another frequent command that causes severe finger cramps. You want to find all the files in the current directory that have a type starting with 'D'. You have to type "XD *.D*". Wouldn't it be nice to have a directory program that automatically wildcarded the file specification. While I was fixing up FINDF to make my new FF, I built that feature into the code. I've been too busy or too lazy to do the same for XD, so instead I added the alias line:

```
D xd $d1$u1:$:1*.$.1* $-1
```

This is a little hard to decipher at a glance because of all the dots and colons and asterisks. But here's how it works. Suppose we are in B4: and enter "D .D /AA" (the option /AA means to include SYS and DIR type files). The parameters in the alias have the following values:

```
$D1 = "B"    $U1 = "4"          current drive and user, since
                                          none given explicitly

$:1 = ""                        no file name given

$.1 = "D"                       file type in first parameter

$-1 = "/AA"                     the tail less the first token ".D"

The command is thus translated by ARUNZ into

        XD B4:*.D* /AA
```

Sometimes it can be nice to allow a command that takes a number of alternative options to run with only the option entered on the command line. I have a read file for MEX that provides automated, menu-based operation on PC-PURSUIT. I could invoke it as "MEX PCP". Instead, I have the alias:

```
PCP mex pcp
```

I also do this with the KMD file transfer commands on my Z-Node, where I define the following aliases:

```
S       kmd s $*
SK      kmd sk $*
SB      kmd sb $*
SBK     kmd sbk $*
SP      kmd sp $*
SPK     kmd spk $*
R       kmd r $*
RP      kmd rp $*
RB      kmd rb
RP      kmd rp $*
L       kmd l $*
LK      kmd lk $*
```

This way the user can skip typing "KMD". Actually, these aliases each contain numerous other synonyms as well. The 'S' alias, for example, includes "SEND", "DOWN", and "DOWNLOAD" as well. The cost in terms of disk space to add all these aliases is so small that I let my enthusiasm and imagination run wild. Note, however, that with the above aliases defined, the RCP should not have the 'R' (reset) and 'SP' (space) commands, since they will take precedence over the alias. I changed the names of these commands to 'RES' and 'SPAC'. The remote user has no reason to use them anyway.

There are, of course, many really complicated sequences of commands (editing, assembling, and linking files, for example) that can very nicely be performed by aliases. Those are fairly obvious, and I have described quite a few in previous columns. I won't give any more examples here, but I will describe two special applications where ARUNZ aliases cut down a complex process to simple proportions. The first is automation of the get-poke-go technique pioneered by Bruce Morgen.

Automated GET-POKE-GO

Here the alias does more than just save typing—it remembers the addresses that have to be poked, something you probably can't do. I will illustrate it with an intriguing example that is sort of recursive.

Suppose ARUNZ is the extended command processor, has been renamed CMDRUN.COM, and is set to get its ALIAS.CMD file from the root directory. Next, suppose you also want to be able to invoke it manually and have it, in that case, look for its ALIAS.CMD file along the entire path, including the current directory. Suppose, furthermore, that CMDRUN.COM is a type-3 program that loads and runs at address 8000H.

By inspecting CMDRUN.COM, we find that we have to poke a 0 at offset 1CH (address 801CH) to turn off the ROOT configuration option and an FFH at offset 24H (address 8024H) to turn on the SCANCUR option. If we are to make manual invocations using the alias name 'RUN', we can put the following line in the ALIAS.CMD file in the root directory, where the unpoked CMDRUN.COM will find it:

```
RUN get 8000 cmdrun.com;poke 801c 0;poke 8024 ff;jump 8000 $*
```

I particularly chose this example because it illustrates the slightly more advanced version of GET-POKE-GO called GET-POKE-JUMP. One word of caution. This technique will only work under ZCPR33. BGii version 1.13 is very close to ZCPR33, but it still handles the JUMP command the way ZCPR30 did, and it cannot use JUMP when a command tail is processed.

I will now describe two very special operations that can be performed very nicely with ARUNZ aliases: recursion and repetition.

Special Recursion Aliases

The following pair of aliases (more or less) that implement Dreas Nielsen's recursion technique were described in my column in issue 28. They allow one to execute a single command recursively. With each cycle the user will be asked if he wants to continue. So long as the answer is yes, the command will be executed repeatedly. Upon a negative reply, the recursive sequence will terminate, and any pending commands will execute.

The alias that the user invokes can be called "REC.URSE" so that it can be invoked with a simple 'REC'. It contains the following sequence of commands:

```
if nu $1
  echo;echo %<  s%>yntax: %<$0 cmdname [parameters]^j
else
  /recurse2 $*
fi
```

If invoked without at least a command name, this alias echoes a syntax message to the screen. Otherwise it invokes the second alias RECURSE2. The leading slash speeds things up by signaling the ZCPR33 command processor that it should go directly to the extended command processor. If you are using BackGrounder-ii (version 1.13), the slash should be replaced by a space (the alias will then work with BGii or Z33). If you are using ZCPR30, don't use either; a space won't do you any good, and a slash will cause the command to fail.

The alias that does the real recursion (RECURSE2) has the following sequence of commands:

```
fi
$*
if in r%>un %<"$*" %>again?
  /$0 $*
```

If the user answers the 'run again' query affirmatively, RECURSE2 will be invoked again. By using '$0' instead of 'RECURSE2' the script will work even if we later change its name.

Special REPEAT Alias

Here is a simple special alias that will allow a command that takes a single argument (token) to be repeated over a whole list of arguments separated by spaces (not commas). The name of the alias is "REP.EAT" so that it can be invoked with a brief 'REP'. The script contains the following commands:

```
$zxif
if ~nu $2
  echo $1 $2
  $1 $2
fi
if ~nu $3
  /$0 $1 $-2
fi
```

The '$z' in the first line declares the alias to be in recursive mode (any pending commands in the multiple command line buffer are dropped when this alias executes), and 'xif' clears the flow state. Invoked as:

REPEAT CMDNAME ARG1 ARG2 ARG3

for example, interpretation of the script the first time through results in the following commands:

```
xif
if ~nu arg1
  echo cmdname arg1
  cmdname arg1
fi
if ~nu arg2
  /repeat cmdname arg2 arg3
fi
```

The command line generated ("CMDNAME ARG1") is first echoed to the screen so the user knows what is going on, and then it is run. Since there is a second argument, the alias is reinvoked as "REPEAT CMDNAME ARG2 ARG3". Note that the first argument has been stripped away. After "CMDNAME ARG2" has also been run and stripped from the command, the interpreted command string will be:

```
xif
if ~nu arg3
  echo cmdname arg3
  cmdname arg3
fi
if ~nu
 /repeat cmdname
fi
```

This time the null test in the second IF clause will fail, and the cycle of commands will come to an end.

This form of the REPEAT alias suffers from the problems Dreas Nielsen pointed out (it wipes out any commands following it on the original command line). A rigorous version can be made (adapting Dreas's technique) by making two aliases as follows:

```
REP.EAT
     if nu $2
       echo;echo %<  s%>yntax: %<$0 aliasname arg1 arg2 ...^j
     else
       /repeat2 $*
     fi

REPEAT2
     fi
     $1 $2
     if ~nu $3
       /$0 $1 $-2
```

If there is not at least one argument after the name of the command, a syntax message is given. Otherwise a series of operations using REPEAT2 begins in which the command is executed on the first argument, and then REPEAT2 is reinvoked with the same command name but with one argument stripped from the list of arguments. Note that the parameter $-2 is used. The first parameter (the command verb) is given explicitly as $1. "$-2" strips away the verb and the argument that has already been processed. The expression "$1 $-2" allows one to strip out the second token. Similarly, "$1 $2 $-3" would strip out the third token. "$1 $-3" would strip out the second and third tokens, leaving the first one intact and moving the remaining tokens down by two.

Configuring ARUNZ

There are several configuration options that allow the user to tailor the way ARUNZ operates. The COM file is designed to make it easy to patch in new values for most of the options using a program like ZPATCH.

Execution Address for ARUNZ

ARUNZ is written as a type-3 ZCPR33 program. In other words, it can automatically be loaded to and execute at an address other than 100H. In this way, its invocation as an extended command processor can leave most of the TPA (transient program area) unaffected by its operation. In the LBR file posted on RASs there are generally two versions of ARUNZ, one designed to run at 100H (and usable in ZCPR30 systems) and one designed to run at 8000H. Sometimes there are also REL files that the user can link with the ZCPR libraries to run at any desired address.

Display Control

There are two bytes just after the standard ZCPR3 header at offset 0DH in the COM file (just before the string "REG") that control the display of messages to the user during operation of ARUNZ. The first byte applies when ARUNZ has been invoked under ZCPR33 as an extended command processor; the second applies to manual invocation (or any use under ZCPR30).

Each bit of these two bytes could control one display feature. At present, only six of the bits are used. Setting a bit causes the message associated with the bit to be displayed; resetting the bit supresses the display of the corresponding message.

The least significant bit (bit 0) affects the program signon message. The usual setting is 'off' for ECP invocations and 'on' for manual invocations. Bit 1 affects the display of a message of the form;

Running alias "XXX"

This message is normally displayed only for manual invocations of ARUNZ.

Bit 2 controls the display of the "ALIAS.CMD file not found" message. This message should generally be enabled, since it will not appear unless something has unexpectedly gone wrong, and you might as well know about it.

Bit 3 controls the display of a message of the form:

Alias "XXX" not found

This message is normally turned on for manual invocations only. When the alias is not found by ARUNZ operating as a ZCPR33 ECP, control is turned over to the error handler, and there is no need for such a message. The message can alternatively be generated, in whatever form the user desires, using a default alias as described earlier. In that case, however, the message will appear for ECP as well as manual invocations.

Bits 4 and 5 apply only when ARUNZ has been invoked as an extended command processor, and they were included as a debugging aid while I was first developing ARUNZ. Both are normally turned off. If bit 4 is set, ARUNZ will display the message "extended command processor error" if it could not process the alias during an ECP invocation. Bit 5 controls a message of the form "shell invocation error". It is possible (though very tricky and not recommended) for an alias to serve as a shell. If ARUNZ fails to find an alias when invoked as a shell processor, then this message will be displayed if bit 5 is set.

Locating the ALIAS.CMD File

There are several possibilities for how ARUNZ is to go about locating the ALIAS.CMD file. There are four configuration blocks near the beginning of the ARUNZ.COM file; they are marked by text strings "PATH", "ROOT", "SCANCUR", and "DU". If the byte after "PATH" is a zero, then ARUNZ will look in the specific drive and user areas indicated by the two bytes following the string "DU". The first byte is for the drive and has a value of 0 for drive A, 1 for B, and so on. The second byte has the user number (00H to 1FH).

If the byte after the string "PATH" is not zero, then some form of path search will be performed depending on the settings of the bytes after the strings "ROOT" and "SCANCUR". If the byte after "ROOT" is zero, then the entire ZCPR3 path will be searched. If the byte after "SCANCUR" is nonzero, then the currently logged drive and user will be included at the beginning of the path. If the byte after "ROOT" is nonzero, then only the root directory (last directory specified in the path) will be searched, and the byte after "SCANCUR" is ignored.

My general recommendation is to use either the root of the path or a specified DU, especially when ARUNZ is being used as the extended command processor. It can take a great deal of time to search the entire path including the current directory. With ARUNZ as the ECP this will be done every time you make a typing mistake in the entry of a command name, and the extra disk accesses can get quite tedious and annoying.

Use Register for Path Control

There is an alternative way to control the path searching options that can give one the best of all possible worlds. After the string "REG" one can patch in a value of a user register, the value of which will be used to specify the path search options PATH, ROOT, and SCANCUR instead of the fixed configuration bytes described above.

Any one of the full set of 32 ZCPR3 registers can be specified for this function by patching in a value from 00H to 1FH. If any other value is used, the fixed configuration bytes will be used. If a valid register is specified, its contents are interpreted as follows:

```
bit 0   PATH flag (0 = use fixed DU; 1 = use path)
bit 1   ROOT flag (0 = use entire path; 1 = use root only)
bit 2   SCANCUR flag (0 = use path only; 1 = include
                current DU)
```

By changing the value stored in the specified register, one can change the way ARUNZ looks for the ALIAS.CMD file dynamically depending on the circumstances.

Plans for the Future

I don't have much writing stamina left, but I would like to finish with a few comments about developments I would still like to see in ARUNZ. A few were mentioned in the main text above. There is a need for some additional parameters, such as register values in various decimal formats. One also needs more flexible access to the directory specification part of a token. The present parameters only allow extracting a DU reference, and they don't allow any way to tell if an explicit directory is specified. There should be a parameter that returns whatever DU or DIR string (including the colon) is present. If none is present, the parameter should return a null string.

One of the things hampering the additional of more parameters is the arcane form they presently take. I would like to find a much more rational system (and if you have any suggestions, I would love to hear them). I am thinking of something like $S for system file, followed by 'F', 'N', or 'T' and then a number 0..3. Thus $ST2 would read Systemfile-Type-2. Command line tokens might be $T followed by 'D', 'U', 'P', 'F', 'N', or 'T' and then a digit 0..9 or 1..9. The 'P' option (path) would be the DU or DIR prefix, if any, including the colon. Problem: what letter do I use for the named directory or the path without the colon? The logical choices 'N' and 'D' are already used. Maybe I have to go to four letters: $T for token, followed by 'D' for directory part or 'F' for file part. The 'D' could be followed by various letters (again, I am not sure what to use for all of them) to indicate:

1) the equivalent drive or default if none specified.
2) the equivalent drive or null string if none specified.
3) the same two possibilities for the user number.
4) the equivalent or given named directory (but what if the directory has no name).
5) the whole directory prefix as given either including or not including the colon.

Similarly, the 'F' option could be followed by a letter to indicate the whole filename, the name only, or the type only. As you can see, it is not easy to identify all the things one might need and find a rational way to express them all.

It would be nice to have prompted input where the user's input could be used in more than one place in the command line. User input would have to be assigned to temporary parameters ($U1, $U2, and so on). Perhaps there should be the possibility of specifying default values for command line tokens when they are not actually given on the command line (as in ZEX). It might also be useful to be able to pull apart a token that is a list of items separated by commas.

ARUNZ could use better error reporting for badly formed scripts. At present one just gets a message that there was an error in the script, but there is no indication of what or where the error was. Ideally, the interpreted line should be echoed to the screen with a pointer to the offending symbol (NZEX has this).

There should be an option to have ARUNZ vector control to the ZCPR3 error handler whenever it cannot resolve the alias or when there is a defect in the script. At present, chaining to the error handler only occurs when ARUNZ has been invoked as an ECP.

An intriguing possibility is to allow alias name elements to be regular expressions in the Unix (or JetFind) sense. Then one could give an alias name like "[XS]DIR" to match either XDIR or SDIR. Perhaps there could be a correspondence established between non-unique expressions and a parameter symbol in the script. Then all my KMD aliases might be simpler:

```
S[P]*[K]* kmd s$x1$x2 $*
```

The name would read as follows: 'S' followed by zero or more occurrences of 'P' followed by zero or more occurrences of 'K'. The parameter $X1, for example, would be the first regular expression, i.e., the 'P' if present or null if not. This is fun to think about, but I am not at all sure that it would really be worth the trouble to use or to code for. Any comments?

It would also be nice to provide Dreas Nielsen's RESOLVE facility directly in ARUNZ aliases. These would use the percent character ('%') as a lead-in. Any symbol enclosed in percent signs would be interpreted as a shell variable name, and its value from the shell file would be substituted for it in the command line. The parameter '$%' would be used to enter a real percent character.

Next Time

As usual, I have written much more than planned but not covered all the subjects planned. I really wanted to discuss shells in more detail, particularly after the fiasco with the way WordStar 4 behaves by trying to be a shell when it should not be. That will have to wait for next time, I am afraid. Also by next time I should be ready to at least begin the discussion of ZEX. ■

The CP/M Corner

by Bob Blum

Patching CP/M to correct problems or add functionality has, for good reason, become practically an art form over the years. Most often patches are applied using one of two forms: altering the existing code "on the fly" from a logic section typically residing in the BIOS, or permanently overlaying existing code and unused memory areas with the desired changes.

Either patching method serves the intended purpose without any clear technical advantage over the other. The deciding factor in favor of a method will probably be based on the availability of the BIOS source code for the target system. If the BIOS source is readily available and the necessary programming tools and talent are handy, then making the change elsewhere seems unwise. Some computer manufacturers, however, do not distribute the BIOS source code making the decision simple.

There are of course conditions to be aware of when using either technique. The most poignant example of what can happen when making indiscriminate patches happened to me several years ago while testing a CCP replacement program. After many hours of tracking a very illusive logic problem it came to my attention that a certain section of memory was being inadvertently altered. I immediately set up a test condition that monitored the memory location and would halt execution of the program as soon as the memory area was again altered.

Finally it happened again, and as desired program execution was stopped, what I found as a result of this exercise still brings color to my cheeks. Many months earlier I had made a BIOS modification to "on the fly" alter the standard CCP code if a particular error condition occurred. The error condition was happening as desired, but unfortunately the CCP code being altered was now different.

Patching over existing code sections can be dangerous as well if the original code is not saved in case it is desired at a later time to back out of the changes.

As an example of both types of patching in the form of a very useful change to the CCP study both figures 1 and 2. Each routine causes user area 0 to be searched for a .COM file in the event it is not found in the current user area.

Please excuse the brevity of this issue's column. Some surgery early in early December and a longer than expected recovery period has put me far behind and my deadline has already passed. ■

FIGURE 1 - Patching method of CCP modification

```
          TITLE     'CCPUSR IF PRIME FILE NOT FOUND SEARCH USER 0'

;PATCH FOR DIGITAL RESEARCH'S CP/M 2.2 CCP

;BY       L. BARKER
;         P.O. BOX 135
;         CHICAGO IL 60690

;THIS PATCH IS TO ALLOW A USER TO ACCESS USER 0
;FILES AS A DEFAULT FOR THE PRIMARY FILE SEARCH
;(ON THE CURRENTLY ACCESSED DISK) IF THE FILE
;CAN NOT BE FOUND IN THE ACTIVE USER AREA.

;    (JUST PATCHING THE JZ AND ADDING 31 BYTES
;OF CODE WILL WORK ONLY IF ALL USER0 .COM FILES
;ARE NO LONGER THAT ONE EXTENT.)

CCP       EQU  0D600H           ;A COMPUPRO VERSION
;; 1024 DSDD W/DUU              T=0 S=12  T=2 S=14
;; CCP    EQU  0A800H           ;OSI C3 CCP ADDRESS
BUF       EQU  0080H
TAP       EQU  0100H

          ;CCP ROUTINE EQUATES

OPENP     EQU  CCP+00D0H
READSQ    EQU  CCP+00F9H
GETUSR    EQU  CCP+0113H
SETUSR    EQU  CCP+0115H
SETDMA    EQU  CCP+01D8H
READOK    EQU  CCP+0701H
BADFIL    EQU  CCP+076BH
BADCCP    EQU  CCP+0771H
FCBCCP    EQU  CCP+07CDH

SYSGEN    EQU  100H             ;FOR THIS SAMPLE
FREESP    EQU  200H             ;TO SHOW CODE SIZE
PATCH2    EQU  0EFC6H

          ORG  SYSGEN           ;WHERE IN IMAGE TO PATCH

PATCH1    EQU  CCP+06DBH
          JMP  CCPFIX           ;<=====  PATCH ==
          LXI  H,TPA

LOOP      EQU  PATCH1+6
          PUSH H                ;ORIG CCP CODE
          XCHG                  ;W/2 JMP PATCHED
          CALL SETDMA
          LXI  D,FCBCCP
          CALL READSQ           ;READ A SEC
          JNZ  EXIT1            ;TILL DONE
          POP  H
          LXI  D,BUF            ;MOVE DMA
          DAD  D                ;ONE UP.
          LXI  D,CCP
          MOV  A,L              ;IF
          SUB  E                ;PGM > CCP-TPA
          MOV  A,H
          SBB  D                ;THEN
          JNC  EXIT2            ;EXIT  <===
          JMP  LOOP             ;ELSE AGAIN
;READOK:
          ORG  FREESP

USERN     EQU  PATCH2           ;SOMEWHERE FREE
          DB   0

;TEST IF OPEN FAILED AND IF USER #>0

CCPFIX    EQU  $-FREESP-USERN
          PUSH PSW
          CALL GETUSR           ;GET  USER #
          LXI  H,USERN          ;AND SAVE IT
          MOV  M,A
          POP  PSW
          JNZ  PATCH1+3
          MOV  A,M
          ORA  A
          JZ   BADFIL           ;IF 0 DONE
          MVI  E,0              ;ELSE TRY
          CALL SETUSR           ;AGAIN
          CALL OPENP            ;W/USER 0
          JNZ  PATCH1+3
          CALL RESET
          JMP  BADFIL

;RESTORE CURRENT USER NUMBER
```

```
RESET       EQU  $-FREESP-USERN
            LDA  USERN
            ORA  A                  ;IF ZERO THEN
            RZ                      ;DON'T BOTHER
            MOV  E,A
            JMP  SETUSR

;FILE WAS READ SUCCESSFULLY

EXIT1       EQU  $-FREESP-USERN
            PUSH PSW
            CALL RESET
            POP  PSW
            JMP  READOK

;PROGRAM SIZE > CCP-TPA

EXIT2       EQU  $-FREESP-USERN
            CALL RESET
            JMP  BADCCP

SIZE        EQU  $-FREESP
```

FIGURE 2 - BIOS modification patching of CCP

```
MSIZE     EQU  56                   ;SIZE OF SYSTEM MEMORY
BIAS      EQU  (MSIZE-20)*1024      ;CALCULATION OF OFFSET
CCP       EQU  3400H-BIAS           ;START ADDR OF CCP
PATCH1    EQU  CCP+389H             ;ADDR OF FIRST PATCH
PATCH2    EQU  CCP+6DCH             ;ADDR OF SECOND PATCH
PATCH3    EQU  CCP+6EDH             ;ADDR OF THIRD PATCH
BIOS      EQU  CCP+1600H            ;START ADDR OF BIOS
BDOS      EQU  CCP+806H             ;START ADDR OF BDOS
          ORG  BIOS                 ;ORIGIN OF BIOS
;*************************************************************
;*                                                           *
;*   NORMAL SYSTEM BIOS PROGRAM STARTS HERE                  *
;*                                                           *
;*************************************************************
***  BIOS THE SAME AS SUPPLIED WITH SYSTEM UP TO THIS ROUTINE ***

GOCPM:    MVI  A,03CH               ;THIS IS THE NORMAL CODE
                                    ;SUPPLIED WITH CP/M
          STA  0
          LXI  H,WBOOTE
          SHLD 1
          STA  5
          LXI  H,BDOS
          SHLD 6
          LXI  B,80H
          CALL SETDMA
          EI
          LDA  CDISK
          MOV  C,A
                                    ;THIS BEGINS THE CODE ADDED
                                    ;TO THE GOCPM PORTION OF CP/M
                                    ;
          LXI  H,CHECK              ;LOAD ADR OF CHECK ROUTINE
          SHLD PATCH2               ;STORE AT PATCH2
          LXI  H,USRRST             ;LOAD ADR OF USRRST ROUTINE
          SHLD PATCH3               ;STORE AT PATCH3
          LXI  H,PROMPT             ;LOAD ADR OF PROMPT ROUTINE
          SHLD PATCH1               ;STORE AT PATCH1
          JMP  CCP                  ;JUMP TO BEGINNING OF CCP
                                    ;
;*************************************************************
;*                                                           *
;*   START NEW ROUTINES AT END OF                            *
;*   EXISTING BIOS PROGRAM                                   *
;*                                                           *
;*************************************************************
                                    ;
BDOSE     EQU  5                    ;ADR OF BDOS ENTRY POINT
                                    ;USED FOR SYSTEM CALLS
OPEN      EQU  CCP+0D0H             ;CALL THIS LOCATION TO
                                    ;RE-INITIALIZE THE SEARCH
                                    ;FILE FOUND IN DIR 0
MFOUND    EQU  CCP+76BH             ;ADR TO RETURN TO IF FILE NOT
                                    ;FOUND
EOF       EQU  CCP+701H             ;JUMP TO THIS ADR AFTER USER #
                                    ;RESTORE OPERATION
RFILE     EQU  CCP+6DEH             ;JUMP TO THIS ADR TO READ FILE
PCHAR     EQU  CCP+8CH              ;CALL THIS LOCATION TO PRINT
                                    ;USER # PROMPT CHARS
CPMPT     EQU  CCP+1D0H             ;JUMP TO THIS LOCATION AFTER
                                    ;PRINTING USER # PROMPT
                                    ;
;*************************************************************
;*                                                           *
;*   ROUTINE TO MODIFY CP/M PROMPT                           *
;*                                                           *
;*************************************************************
;
PROMPT:   PUSH B                    ;SAVE BC AND DE REGS
          PUSH D
          MVI  C,20H                ;SYSTEM CALL 20H (INTERROGATE)
          MVI  E,0FFH               ;USER NUMBER
          CALL BDOSE                ;RETURNS WITH CURRENT USER #
                                    ;IN THE A REG
          CPI  0AH                  ;IS THE USER # > 10?
          JNC  CHAR2                ;IF SO MUST PRINT TWO #'S
PMT1:     ADI  30H                  ;OTHERWISE MAKE ASCII
          CALL PCHAR                ;OUTPUT # TO CONSOLE
EXIT1:    POP  D                    ;RESTORE BC AND DE REGS
          POP  B
          JMP  CPMPT                ;RETURN CONTROL TO CCP @ CPMPT
CHAR2:    SUI  0AH                  ;USER # IS > 10 SO SUBTRACT 10
          PUSH PSW                  ;SAVE RESULT ON STACK
          MVI  A,31H                ;SEND A ASCII 1 TO CONSOLE
          CALL PCHAR
          POP  PSW                  ;RECOVER REMAINDER
          JMP  PMT1                 ;JUMP TO OUTPUT REMAINDER
                                    ;
;*************************************************************
;*                                                           *
;*   ROUTINE TO CHECK 0 FOR FILE                             *
;*                                                           *
;*************************************************************

;
CHECK:    PUSH D                    ;SAVE BC,DE, AND AF REGS
          PUSH B
          PUSH PSW
          MVI  A,0                  ;RESET FLAG
          STA  FLAG
                                    ;FLAG INDICATES THAT USER #
                                    ;CHANGED IF FLAG IS SET
          MVI  E,0FFH               ;INTERROGATE CURRENT USER #
          MVI  C,20H
          CALL BDOSE
          STA  USER                 ;STORE CURRENT USER # IN TEMP
          ORA  A                    ;CHCECK FOR USER 0
          JNZ  NUSERO               ;JUMP IF CURRENT USER # NOT
                                    ;USER 0
          JMP  EXIT                 ;IF CURRENT USER # IS USER 0
                                    ;THEN RETURN CONTROL TO CCP
NUSERO:   MVI  C,20H                ;SET USER # TO USER 0
          MVI  E,0
          CALL BDOSE
          CALL OPEN                 ;RE-INITIATE SEARCH
          JNZ  FOUND                ;IF A REG RETURNED NON 0 THEN
                                    ;FILE WAS FOUND IN USER 0 DIR
          LDA  USER                 ;OTHERWISE FILE WAS NOT FOUND
                                    ;SO RESTORE USER #
          MOV  E,A
          MVI  C,20H
          CALL BDOSE
EXIT:     POP  PSW                  ;RESTORE DE, BC AND AF REGS
          POP  B
          POP  D
          JMP  NFOUND               ;RETURN CONTROL TO CCP @
                                    ;NFOUND
FOUND:    POP  PSW                  ;FILE FOUND IN USER 0 DIR
          POP  B                    ;RESTORE DE, BC AND AF REGS
          POP  D
          MVI  A,1                  ;SET FLAG FOR RESTORE
                                    ;OPERATION AFTER FILE IS
                                    ;LOADED
          STA  FLAG
          JMP  RFILE                ;RETURN CONTROL TO CCP @ RFILE
                                    ;
;*************************************************************
;*                                                           *
;*   ROUTINE TO RESTORE USER # AFTER                         *
;*   LOADING FILE FROM USER 0 DIR                            *
;*                                                           *
;*************************************************************
                                    ;
USRRST:   PUSH D                    ;SAVE DE, BC AND AF REGS
          PUSH B
          PUSH PSW
          LDA  FLAG                 ;CHECK FLAG
          ORA  A
          JZ   RSTR1                ;IF FLAG NOT SET NO RESTORE
                                    ;REQUIRED
          LDA  USER                 ;IF FLAG NON 0 THEN GET USER #
          MOV  E,A                  ;RESTORE USER #
          MVI  A,20H
          CALL BDOSE
          MVI  A,0                  ;RESET FLAG
          STA  FLAG
PSTR1:    POP  PSW                  ;RESTORE DE, BC AND AF REGS
          POP  B
          POP  D
          JMP  EOF                  ;RETURN CONTROL TO CCP @ EOF
FLAG:     DB   0                    ;LOCATION OF FLAG
USER:     DB   0                    ;LOCATION OF TEMP USER #
                                    ;STORAGE
                                    ;USE DB INSTEAD OF DS TO
                                    ;INSURE FLAG AND USER ARE
                                    ;SET TO 0 INITIALLY
LAST:     EQU  $-BIOS               ;NOTE FOR SINGLE DENSITY 8"
                                    ;DISK USERS:
                                    ;IF LAST > 37FH THEN BIOS TOO
                                    ;BIG TO FIT ON SYSTEM TRACKS
                                    ;(TRACKS 0 AND 1 OF DISKETTE)

                                    ;
;*************************************************************
;*                                                           *
;*   LOCATE NORMAL CP/M DIRBUF, ALLOCATION                   *
;*   STORAGE AND CHECK VECTORS AFTER ADDED CODE              *
;*                                                           *
;*************************************************************
          END
```

Z sets you free!

Who we are

Echelon is a unique company, oriented exclusively toward your CP/M-compatible computer. Echelon offers top quality software at extremely low prices; customers are overwhelmed at the amount of software they recieve when buying our products. For example, the Z-Com product comes with approximately 92 utility programs; and our TERM III communications package runs to a full megabyte of files. This is real value for your software dollar.

ZCPR 3.3

Echelon is famous for our operating systems products. ZCPR3, our CP/M enhancement, was written by a software professional who wanted to add features normally found in minicomputer and mainframe operating systems to his home computer. He succeeded wonderfully, and ZCPR3 has become the environment of choice for "power" CP/M-compatible users. Add the fine-tuning and enhancements of the now-available ZCPR 3.3 to the original ZCPR 3.0, and the result is truly flexible modern software technology, surpassing any disk operating system on the market today. Get our catalog for more information - there's four pages of discussion regarding ZCPR3, explaining the benefits available to you by using it.

Z-System

Z-System is Echelon's complete disk operating system, which includes ZCPR3 and ZRDOS. It is a complete 100% compatible replacement for CP/M 2.2. ZRDOS adds even more utility programs, and has the nice feature of no need to warm boot (^C) after changing a disk. Hard disk users can take advantage of ZRDOS "archive" status file handling to make incremental backup fast and easy. Because ZRDOS is written to take full advantage of the Z80, it executes faster than ordinary CP/M and can improve your system's performance by up to 10%.

Installing ZCPR3/Z-System

Echelon offers ZCPR3/Z-System in many different forms. For $49 you get the complete source code to ZCPR3 and the installation files. However, this takes some experience with assembly language programming to get running, as you must perform the installation yourself.

For users who are not qualified in assembly language programming, Echelon offers our "auto-install" products. Z-Com is our 100% complete Z-System which even a monkey can install, because it installs itself. We offer a money-back guarantee if it doesn't install properly on your system. Z-Com includes many interesting utility programs, like UNERASE, MENU, VFILER, and much more.

Echelon also offers "bootable" disks for some CP/M computers, which require absolutely no installation, and are capable of reconfiguration to change ZCPR3's memory requirements. Bootable disks are available for Kaypro Z80 and Morrow MD3 computers.

Z80 Turbo Modula-2

We are proud to offer the finest high-level language programming environment available for CP/M-compatible machines. Our Turbo Modula-2 package was created by a famous language developer, and allows you to create your own programs using the latest technology in computer languages - Modula-2. This package includes full-screen editor, compiler, linker, menu shell, library manager, installation program, module library, the 552 page user's guide, and more. Everything needed to produce useful programs is included.

"Turbo Modula-2 is fast...[Sieve benchmark] runs almost three times as fast as the same program compiled by Turbo Pascal...Turbo Modula-2 is well documented...Turbo's librarian is excellent". - Micro Cornucopia #35

BGii (Backgrounder 2)

BGii adds a new dimension to your Z-System or CP/M 2.2 computer system by creating a "non-concurrent multitasking extension" to your operating system. This means that you can actually have two programs active in your machine, one or both "suspended", and one currently executing. You may then swap back and forth between tasks as you see fit. For example, you can suspend your telecommunications session with a remote computer to compose a message with your full-screen editor. Or suspend your spreadsheet to look up information in your database. This is very handy in an office environment, where constant interruption of your work is to be expected. It's a significant enhancement to Z-System and an enormous enhancement to CP/M.

BGii adds much more than this swap capability. There's a background print spooler, keyboard "macro key" generator, built-in calculator, screen dump, the capability of cutting and pasting text between programs, and a host of other features.

For best results, we recommend BGii be used only on systems with hard disk or RAMdisk.

JetFind

A string search utility is indispensible for people who have built up a large collection of documents. Think of how difficult it could be to find the document to "Mr. Smith" in your collection of 500 files. Unless you have a string search utility, the only option is to examine them manually, one by one.

JetFind is a powerful string search utility which works under any CP/M-compatible operating system. It can search for strings in text files of all sorts - straight ASCII, WordStar, library (.LBR) file members, "squeezed" files, and "crunched" files. JetFind is very smart and very fast, faster than any other string searcher on the market or in the public domain (we know, we tested them).

Software Update Service

We were suprised when sales of our Software Update Service (SUS) subscriptions far exceeded expectations. SUS is intended for our customers who don't have easy access to our Z-Node network of remote access systems. At least nine times per year, we mail a disk of software collected from Z-Node Central to you. This covers non-proprietary programs and files discussed in our Z-NEWS newsletter. You can subscribe for one year, six months, or purchase individual SUS disks.

There's More

We couldn't fit all Echelon has to offer on a single page (you can see how small this typeface is already!). We haven't begun to talk about the many additional software packages and publications we offer. Send in the coupon below and just check the "Requesting Catalog" box for more information.

Item	Name	Price	
1	ZCPR3 Core Installation Package	$49.00	(3 disks)
2	ZCPR3 Utilities Package	$89.00	(10 disks)
5	Z-Com (Auto-Install Complete Z-System)	$119.00	(5 disks) *
6	Z-Com "Bare Minimum"	$69.95	(1 disks)
10	BGii Backgrounder 2	$75.00	(2 disks)
12	PUBLIC ZRDOS Plus (by itself)	$59.50	(1 disk)
13	Kaypro Z-System Bootable Disk	$69.95	(3 disks)
14	Morrow MD3 Z-System Bootable Disk	$69.95	(2 disks)
16	QUICK-TASK Realtime Executive	$249.00	(3 disks)
17	DateStamper file time/date stamping	$49.95	(1 disk)
18	Software Update Service	$85.00	(1 yr sub)
20	ZAS/ZLINK Macro Assembler and Linker	$69.00	(1 disk)
21	ZDM Debugger for 8080/Z80/HD64180 CPU's	$50.00	(1 disk)
22	Translators for Assembler Source code	$51.00	(1 disk)
23	REVAS3/4 Disassembler	$90.00	(1 disk)
24	Special Items 20 through 23	$169.00	(4 disks)
25	DSD-80 Full Screen Debugger	$129.95	(1 disk)
27	The Libraries SYSLIB, Z3LIB, and VLIB	$99.00	(8 disks)
28	Graphics and Windows Libraries	$49.00	(1 disk)
29	Special Items 27, 28, and 82	$149.00	(9 disks)
30	Z80 Turbo Modula-2 Language System	$89.95	(1 disk)
40	Input/Output Recorder IOP (I/OR)	$39.95	(1 disk)
41	Background Printer IOP (BPrinter)	$39.95	(1 disk)
44	NuKey Key Redefiner IOP	$39.95	(1 disk)
45	Special Items 40 through 44	$89.95	(3 disks)
60	DISCAT Disk cataloging system	$39.99	(1 disk)
61	TERM3 Communications System	$99.00	(6 disks)
64	Z-Msg Message Handling System	$99.00	(1 disk)
66	JetFind String Search Utility	$49.95	(1 disk)
81	ZCPR3: The Manual bound, 350 pages	$19.95	
82	ZCPR3: The Libraries 310 pages	$29.95	
83	Z-NEWS Newsletter 1 yr subscription	$24.00	
84	ZCPR3 and IOPs 50 pages	$9.95	
85	ZRDOS Programmer's Manual 35 pages	$8.95	
88	Z-System User's Guide 80 page tutorial	$14.95	

* Includes ZCPR3: The Manual

Echelon, Inc.

P.O. Box 705001-800
South Lake Tahoe, CA 95705
(916) 577-1105

NAME ______________________________
ADDRESS ______________________________
____
TELEPHONE ______________ DISK FORMAT ________
☐ REQUESTING CATALOG

ORDER FORM

Payment to be made by:
☐ Cash
☐ Check
☐ Money Order
☐ UPS COD
☐ Mastercard/Visa:
____
Exp. Date ______________

California residents add 7% sales tax. Add $4.00 shipping/handling in North America, actual cost elsewhere

ITEM	PRICE
________	________
________	________
________	________
Subtotal	________
Sales Tax	________
Shipping/Handling	________
Total	________

Remote

Designing a Remote System Program

by Al J. Szymanski

The program that I am presenting here came about because of a need on my part to move files from two incompatible systems. This program may also become the core for a fully featured remote system driver. I have a Tiny Giant 68000 with K-OS and I love it. I also have my trusty CP/M system which I use as a development system. Herein lies my first headache, format incompatibility. The K-OS uses the MSDOS/IBM format for disks and I cannot utilize that format with my CP/M system. So I wrote this program to allow me to drive the K-OS system as a remote from the CP/M system. Specifically, I now can develop code on the Z80 and then ship it to the 68000 to assemble and run (More on WHY later). Additionally, in order to send TCJ the code and articles on one disk I had to be able to ship code from the 68000 back to the Z80. I generally try to write straightforward code in the sense of making tools, if it works for me then it's OK. This is the first effort I've made to make code as bombproof and friendly as I could. I also realize that we all hate to re-invent the wheel, thus the reason for my sharing this code.

I will discuss each part of the program, as it appears in the listing. HTPL is a Forth-like programming language that has been covered in previous articles. It uses the stack for the evaluation of variables. Since it is very sensitive to any garbage left on the stack, your code must be very clean. I felt that it was fairly easy to create simple tools with HTPL, also it was the only language available on the 68000 system. As I am a die-hard C programmer, it is my intent to port C over to the K-OS environment.

First up are the variable declarations, **AUX:** being the filename of the auxillary port. **Bbuf** is the string input buffer, **Rwbuf** is a buffer for reading and writing to and from the disk. **Abuf** is a small character buffer. **Recl** and **rec2** are the received record number and its complement, used to verify logical sequencing of the records. **Try** is the counter for the

```
( remote. v1.0 in HTPL by Al J. Szymanski 11/87)
root
byte menu = "MENU: R receive, S send, V view, Q quit :>" ;
byte smsg = "Send " ;
byte rmsg = "Receive " ;
byte vmsg = "View " ;
byte cmsg = "Command not implemented." ;
byte qmsg = "Quitting Remote." ;
byte fmsg = "Filename.ext ? " ;
byte auxname = "AUX:" ;
byte bbuf [ 128 ] ;
byte rwbuf [ 512 ] ;
byte abuf [ 4 ] ;
byte recl rec2 try response checksum ;
word lastrec status flchan auxchan ;
word time = 2500 ;   (this is a MAGIC number, this works for 1200 baud)
word pblock [ 20 ] ;
long bufptr ;

program
     openaux
     "Remote.V 1.0 - Ready" writeln
     while auxgetc 3 <> do
          "Remote.V 1.0 - waiting for  control^C" auxputs 13 auxputc
     end
     repeat #menu auxputs auxgetc toupper auxcrlf
     case     [ 'R' ] receive
          [ 'S' ] send
          [ 'V' ] view
          [ 'Q' ] quit
          else dontknow
     end
     false until
end

(**********PRIMARY ROUTINES**********)

proc receive
     #rmsg auxputs getfilec #rwbuf !bufptr 0 !lastrec
     while auxstat =0 do 21 auxputc @time wait end
     repeat
          auxin dup
          if 1 = then
               6 !response
               auxgetc !recl auxgetc not !rec2
                        0 dup !checksum
               while dup 128 <> do
                    dup auxgetc dup @checksum + !checksum
                    @bufptr rot + !1 +1
               end drop (128)
               @checksum $00FF and auxgetc $00FF and
               if <> then 21 !response end
               @recl @rec2
               if <> then 21 !response end
               @response 6 if = then awrite end
               @status $0004 and $0004
               if = then 24 !response end
               @lastrec +1 $00FF and @recl $00FF and
               if <> then 24 !response end
               @recl $00FF and !lastrec
               @response auxputc
               @response 24 if = then return end
          end
     4 = until.
     6 auxputc aclose auxcrlf
end

proc send
     #smsg auxputs getfileo
     0 !recl while auxgetc 21 <> do end
```

attempts made at sending a record, 5 is the current limit. **Response** is the variable used to hold the response to make after receiving a 128 byte packet. It may be either: NAK or 21d , ACK or 6d , CAN or 24d. These are the codes used in the xmodem or Christensen protocol as handshakes. **Checksum** is the sum of the 128 bytes in the packet mod 256. **Lastrec** is the number of the last record received, used to verify that the record that was just received is the next in order. **Status** holds the contents of the status word from the last operating system call made. **Fichan** and **Auxchan** are the file channels or descriptors for the currently open file and for the auxillary port. K-OS treats all files and devices as channels. The word **time** is a variable that was needed to slow down the process of handshaking. (Probably half of the bugs I encountered while working this code out, were due to incoming bytes being stored in the character queue on both machines. This meant that there were occasionally garbage characters waiting and being interpreted as handshakes. This value came about by testing empirically as opposed to calculation and it works for 1200 baud, I don't know what would work for 300). **Pblock** is the structure for all of the operating system calls. **Bufptr** is a long pointer to a byte in the **wrbuf**.

The next block of code is the core of the program. It proceeds as follows: (line 21) open up the channel to the auxillary port, (line 22) send to the 68000 terminal a message that the program is up and running, it's time to switch over to the CP/M machine, (lines 23-24) wait in a loop until a character comes in, assess it for being a ^C, if it is not, send a message asking for the ^C. This was done to clear out the queue. Next enter a large repeat forever loop (lines 26-33) which sends out the menu and waits for a selection. The case evaluates the choice and branches to a routine, with a default at dontknow to bullet-proof the code. The only way out of the code at this level is to enter a 'Q' to quit the program.

Next up are the 5 primary routines; **receive, send, view, quit** and **dontknow**. The basis for what is going on in **send** and **receive** is best described in the article by Donald Krantz, "Christensen Protocols in C," *DR. DOBBS JOURNAL* (#104 June 1985 pp. 66). It is the clearest presentation of the xmodem protocols I have ever read.

Receive works this way: (line 39) it asks for the filename.ext to create and then does so, then starts sending 21d's (NAK)

```
(  71)        repeat
(  72)             aread128 0 !try @recl +1 !recl
(  73)             repeat
(  74)                  1 auxputc 0 dup !checksum
(  75)                  @recl $00FF and dup auxputc not auxputc
(  76)                  while dup 128 <> do
(  77)                       dup #rwbuf + @1 dup
(  78)                       @checksum + !checksum
(  79)                       auxputc +1
(  80)                  end drop
(  81)                  @checksum $00FF and auxputc
(  82)                  @try +1 dup !try if 5 = then errorout end
(  83)             auxgetc dup 24
(  84)             if = then drop aclose return end
(  85)             6 = until
(  86)        @status $0008 and <>0 until
(  87)        4 auxputc
(  88)        aclose
(  89)   end
(  90)
(  91)   proc view
(  92)        #vmsg auxputs getfileo auxcrlf @time wait 0 !status
(  93)        while @status $0008 and =0 do
(  94)             aread1 dup auxputc
(  95)             if 26 = then
(  96)                  aclose auxcrlf return
(  97)             end
(  98)             if auxstat 1 = then
(  99)                  auxgetc
( 100)                  if 3 = then
( 101)                       aclose auxcrlf return
( 102)                  end
( 103)             end
( 104)        end
( 105)   end
( 106)
( 107)   proc quit
( 108)        #qmsg auxwriteln  exit end
( 109)
( 110)   proc dontknow
( 111)        #cmsg auxwriteln end
( 112)
( 113)   (**********O.S. CALLS**********)
( 114)
( 115)   proc openaux
( 116)        #pblock 5 over !2  0 over +2 !2
( 117)        0 over +4 !2 #auxname over 6 + !4
( 118)        trap     #pblock +4 @2 !auxchan
( 119)   end
( 120)
( 121)   proc auxstat
( 122)        #pblock 1 over !2 0 over +2 !2
( 123)        @auxchan over 4 + !2
( 124)        trap #pblock 2 + @2 1 and (returns 1 if char waiting else 0)
( 125)   end
( 126)
( 127)   proc auxin
( 128)        #pblock 2 over !2 0 over +2 !2
( 129)        @auxchan over +4 !2 1 over 6 + !2
( 130)        0 over 8 + !2 #abuf over 10 + !4
( 131)        trap @abuf      (returns the char on stack)
( 132)   end
( 133)
( 134)   proc auxputc
( 135)        !abuf #pblock 3 over !2 0 over +2 !2
( 136)        @auxchan over +4 !2 1 over 6 + !2
( 137)        0 over 8 + !2 #abuf over 10 + !4
( 138)        trap
( 139)   end
( 140)
( 141)   proc acreate
( 142)        #pblock 6 over !2 0 over +2 !2
( 143)        swap over +4 !4 0 over 8 + !4
( 144)        trap #pblock +2 @2 !status
( 145)   end
( 146)
( 147)   proc aopen
( 148)        #pblock 5 over !2 0 over +2 !2
( 149)        0 over +4 !2 swap over 6 + !4
( 150)        trap #pblock +4 @2 !fichan
( 151)        #pblock +2 @2 !status
( 152)   end
( 153)
( 154)   proc aclose
( 155)        #pblock 8 over !2 0 over +2 !2
( 156)        @fichan over +4 !2
( 157)        trap #pblock +2 @2 !status
( 158)   end
( 159)
( 160)   proc awrite
```

```
( 161)       #pblock 3 over !2 0 over +2 !2
( 162)       @fichan over +4 !2 128 over 6 + !2
( 163)       0 over 8 + !2 @bufptr over 10 + !4
( 164)       trap #pblock +2 @2 !status
( 165)  end
( 166)
( 167)  proc aread1
( 168)       #pblock 2 over !2 0 over +2 !2
( 169)       @fichan over +4 !2 1 over 6 + !2
( 170)       0 over 8 + !2 #rwbuf over 10 + !4
( 171)       trap #pblock +2 @2 !status
( 172)       #rwbuf @1      (return read char on stack)
( 173)  end
( 174)
( 175)  proc aread128
( 176)       #pblock 2 over !2 0 over +2 !2
( 177)       @fichan over +4 !2 128 over 6 + !2
( 178)       0 over 8 + !2 #rwbuf over 10 + !4
( 179)       trap #pblock +2 @2 !status
( 180)  end
( 181)
( 182)  proc wait   (expects time on stack)
( 183)       #pblock 53 over !2 0 over +2 !2
( 184)       swap over +4 !4 trap
( 185)  end
( 186)
( 187)  (**********UTILITY ROUTINES**********)
( 188)
( 189)  proc returnstat
( 190)       @status
( 191)       if $8000 and $8000 = then
( 192)            "processed " auxputs
( 193)       end
( 194)       @status
( 195)       if $0004 and $0004 = then
( 196)            "unsucessfully." auxputs
( 197)       else "sucessfully." auxputs
( 198)       end
( 199)  end
( 200)
( 201)  proc crlf
( 202)       13 putc 10 putc end
( 203)
( 204)  proc auxcrlf
( 205)       13 auxputc 10 auxputc end
( 206)
( 207)  proc writeln
( 208)       sprint crlf end
( 209)
( 210)  proc auxwriteln
( 211)       auxputs auxcrlf end
( 212)
( 213)  proc auxgetc
( 214)       while auxstat =0 do end auxin end
( 215)
( 216)  proc auxgets
( 217)       while auxgetc dup <>0
( 218)            do
( 219)                 case
( 220)                 [ 17 ] 30 exit
( 221)                 [ 13 ] drop 0 swap !1 return
( 222)                 [ 8 ] auxputc -1 32 auxputc 8 auxputc 0
( 223)                 else dup auxputc over !1 +1 0
( 224)                 end
( 225)       end
( 226)  end
( 227)
( 228)  proc auxputs
( 229)       while dup @1 dup <>0 do auxputc +1 end
( 230)       drop (pointer to end of string)
( 231)  end
( 232)
( 233)  proc errorout
( 234)       24 dup auxputc auxputc exit end
( 235)
( 236)  proc getfileo
( 237)       filline #bbuf dup auxgets auxcrlf aopen returnstat end
( 238)
( 239)  proc getfilec
( 240)       filline #bbuf dup dup auxgets auxcrlf acreate aopen returnstat end
( 241)
( 242)  proc filline
( 243)       #fmsg auxputs end
( 244)
( 245)  proc toupper
( 246)       dup if 96 > then 32 - end end
( 247)
( 248)  end
( 249)  end
```

until any response is made, then it enters a large loop (lines 41-64) which begins by getting the response and evaluating it for being a 1 or SOH (start of heading), a 4 or EOT (end of text), all other responses being treated as junk. If the response is a SOH it sets up its own response to be a 6 or ACK (acknowledge). Then it gets the next incoming byte which should be the logical record followed by the complement of the logical record. It then enters a small loop (lines 47-50) that just gets the next 128 bytes from the stream and puts them into memory, while calculating the checksum for the record. The byte that follows the data is the sent checksum, which should match our calculated one, if not we change our response to NAK (line 52). It then evaluates the sent record and its complement for errors, again changing the response on error (line 54). If by now no change has been made in the response, it writes the 128 bytes to disk. An evaluation is then made to see if the write went OK, if not the response is changed to 24d or CAN (cancel) which puts an immediate end to the data exchange for both machines. Lastly an evaluation is made to be sure that the record we just got was in fact the next record we should have gotten, if it is not, again we set up to abort the exchange. Finally, the response is sent to the sending machine and exiting if it was the CAN byte. If the first byte sent for the next block was the EOT we have reached the end of the loop and can close the new file and return to the menu.

Send is a much simpler routine. As there can be no transmission errors, all that send needs to do is calculate the checksum and send it and the record numbers when it needs to. First (line 69) asks for the filename.ext to send—opens it and reports status. Then (line 70) waits in a loop until a NAK is received. The user must make sure that checksum mode is set on the receiving machine as the sync byte for CRC mode is 'C', and that is ignored by this routine, (this could cause potential lockup). Once the routine has received the sync byte, it enters a large loop (lines 71-86) in which 128 bytes are sent. **Aread128** is an O.S. call which reads 128 bytes into a buffer. The record count is incremented. A smaller loop (lines 73-85) is then entered. This loop first sends the 1 or SOH byte, starting the transmission block. It then clears the checksum. Next it gets the current record, makes sure it is mod 256 and sends it and its complement (line 75). The inside loop (lines 76-80) actually does the transmission of the 128 bytes of data, calculating the checksum along the way.

```
Fig. 1          (address of pblock) top of stack
                (value of command )
                (address of pblock)
```

```
                                      word width
Fig. 2    address of pblock ==>           3          command value
                                          0          status word
                                    file  chan.  #   from open
                                         128         number of bytes
                                          0          number written
                           |buffer address of source data|  long data
```

The checksum is then sent, again being corrected for mod 256. The try count is incremented and evaluated against bounds, exiting if it exceeds the limit. The next line (line 83) gets the response from the receiving machine, evaluating it for being 24 or CAN, exiting if it is. This meant that the receiving machine found a non-recoverable error. The routine looks for a 6 or ACK as a response, meaning that the record was accepted correctly. The next line (line 86) checks the status word from this last read for the end of the file, and as long as it is not, returns to the top of the loop. If it were the end of the file, a 4 or EOF is sent to complete the transfer, close the file and return to the menu.

View is even simpler than **Send** as no calculations are made, the data is just sent to the receiver's screen. The code is straightforward, however note the check for ^C in the loop (lines 100-101) to cancel the display of the file. **Quit** and **Dontknow** are unremarkable routines.

The next group of routines are the actual calls made to the K-OS operating system. A parameter block is used to handle all of the pointers and values used in an O.S. call. The first word in the block is the actual command code—usually followed by a status word. One of the beauties of this type of arrangement is that you can make as many parameters blocks as you might need and place them anywhere in memory. Pre-loading of parameter blocks and just issuing the trap call at the time of need can save a great deal of time for critical operations. I'll describe just one of the calls for example sake, **Awrite: #pblock** gets the address of the call buffer and places it on the stack, 3 puts the number 3 onto the stack above the address, 'over' is the HTPL macro word that takes the next to the top item on the stack and makes a copy of it and places the copy onto the top of the stack (see Figure 1). The '!2' means: take the top item—treat it as an address and put the next item down into that address, here (line 161) it means put the value 3 into the address of pblock. In doing this, the top two items are removed from the stack, leaving only the original copy of the address. The code proceeds similarly until the '+2' which adds 2 to the top item on the stack, here (line 161) the address of the pblock, offsetting the pointer by one word. It continues until the '@fichan' which means: put onto the stack the item found in the variable **fichan**. By the time the word 'trap' is reached the stack has only the address of the pblock on it, and trap performs the O.S. call. The parameter block looks like Figure 2 before the call is made. After the call is made there is nothing left on the stack from this routine, so we have to replace the address of the parameter block onto the stack to get access to the status word. Then we get the word and store it in the variable status.

The final group of routines are the utility routines which allow for the byte by byte exchange through the auxilliary port and with the 68000 screen. Included in this group is the routine **Returnstat**, which evaluates the status word left from the last operation made and displays the information on the host machine. **Auxgets** allows for inline correction through the auxilliary port while getting a string from the host. **Toupper** does have one quirk in that if the characters '{' through '~' are used it will make them unusable, or at best treat them as the control characters ` [ through ^ ` .

I have used this program now for about a month to make a cross compiler to port a version of C onto the 68000 to run under K-OS. There are a few changes that I would suggest be made. One is to allow the host to view the directory of the 68000 machine and eventually give the host full command level capabilities, even to writing a version of BYE for a full remote operating system. As far as my version of C, I have the cross compiler up and running and have most of the 68000 run time library done. I owe credit to the fine folks at Hawthorne Technology. They are only 40 miles up the road from me and have helped me a lot.

I plan to write a few more articles about the Tiny Giant and on the C compiler I'm working on. That's all for now folks. ■

Reader's Feedback

(Continued from page 5)

natural for combining high-level language with various assembler routines.

What have I been doing? Well, a while back, I indicated that I was building a linear supply for a second SB180. It's almost done and I intend to write up an article on how I designed and built it. Perhaps it will be good enough for TCJ to publish.

I've also been doing a bit of PC Board design on the Mac using MacDraw and some templates that I've designed on my own. Be glad to share that experience as well if your readers might be interested.

T.M.

Editor's Note: We'll be looking forward to power supply article, and encourage the readers to let us know about their interest in PC Board design on the Mac.

Hardware Control

I'm using MTU 130, Mac+, Mac SE, Apple IIgs, Apple IIe, and HP Vectra.

Most of the effort is using the above for data collection and some number crunching. I primarily use True Basic and C for programs, and 6502 assembly for some speed in the Apple II.

I would like a good tutorial on 68000 assembly, and also on FORTH (it seems to me to be a nice language but I haven't sat down to learn it). I also really enjoy articles on hardware control, stepper motors, ADCs, DADs, etc.

D.M.

Ripe Thinking

I'm using PCTECH X16B 10MHz with new OMTI 3520 CCS, controller of two different drives, ST 225 and Minis 3650. Earl Hinrichs software is outstanding.

I used to use (before my desert house in 29Palms was burglarized) in addition to the X16B, a CP/M-86/MS DOS environment: FALCO TS1 terminal, Slicer with Shugart 860-2 and two Mitsi 4853s, housed in a Ferguson BB cabinet with Ferguson UPS. Damnation! The first computer I built, ripped off by someone who didn't take the manuals with the system.

Next quarter, I plan to add a TinyGiant 68000. 68000 is the way to go. I don't like DOS or segmented 86. DOS is a real challenge to learn as a first machine, but when you buy computers from PCTECH and Slicer, you get fantastic support that makes the effort worthwhile.

TCJ is more than a breath of fresh air, it's a perspective, e.g. the editorial with emphasis on real time programming.

I'd like to see articles on cross assemblers, like cross assembling 68000 code on the 8086, vice versa, etc. I've been wondering about relatively cheap cross assemblers such as Austin Codework's $25 A68000.

Also interested in new Zilog Z280, Transputers, NS32X32, digital image progessing with NEC uPD7281, and GSP's like TI's 34010.

Concerning the 32X32 and Job's NEXT machine, and other CPUs they're considering, I somethimes think it should be called WHEN Corporation. Facetiousness aside, I am intrigued what the impact of a UNIX machine will have, including the shock of the retail price of the machine.

I think Don Lancaster did hit one nail on the head. when I paraphrase him from "Ask the Guru" in '85/'86: The Macintosh has a fascist operating system — it forces you to be user friendly.

One of the greatest technical BBSs I've used is Trevor Marshalls 1000 Oaks at 805-493-1495.

Turbo C (the only MS DOS C compiler I have, don't know about other ones) has a good feature with the ability to generate symbolic files in command-line version environment that are compatible with MASM's .SYMDEB — those two switches '-y' and '-m' make it really fun to step through executable files.

Saw a demonstration of Tektronix's 3-d color terminal — you put on polarized lenses and a LCD shutter in front of display, and its software makes basic objects (wire frames in this case) pop out of the terminal. Nice toy at $40,000,but like much technology, a matter of time (decade or so) to have a personal 3-D graphic environment, and speaking of that, holographic environments like in debugging — heap's on my left, stack's on my right, registers straight ahead. How long will Von Neumannism survive?

Thanks for TCJ, every issue is for ripe reading/thinking.

R.S.

32-Bit

I use CP/M Z80 S100 and single board, plus UNIX, VAX, MICROVAX, Sun, etc. (college is such fun, eh?).

I would really like to see more hardware projects — especially in the area of 32-Bit single board computers. I am particularly interested in finding out more about the Zilog Z80,000 32-Bit micro-mainframe. How about someone out there making a workstation (Berkeley UNIX based, of course) based on this chip?

R.A.

SB180

I am running a Micromint SB180, with the hardware mods to enable DTR to my Wyse 30 termainal, and four floppies (2 DSDD and 2 DSQD, all TEAC), as well as the 9.216 MHz upgrade and XBIOS. In other words, I have taken my SB180 nearly, but not quite, as far as it will go. Next step SCSI interface and, hopefully, a 2-4 meg RAM disk. (I really don't want to go Hard Drive, though I might end up doing that. Afraid of reliability problems — probably unjustified, however.)

I'd like to see: 1) SCSI, 2) Solid state "drives" for CP/M or Z, 3) Advanced CPU (Z800, Z280, etc), 4) Interface basics — computer with drives, terminals, DMA/keyboard/monitor vs. terminal, modem.

J.B.

Back Issues Available:

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

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

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

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

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

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

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

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

Issue Number 17:
- Poor Man's Distributed Processing
- BASE: Part Five
- FAX-64: Facsimile Pictures on a Micro
- The Computer Corner

Interfacing Tips & Troubles: Memory Mapped I/O on the ZX81

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
- The Computer Corner

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
- The Computer Corner

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
- The Computer Corner

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
- The Computer Corner

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

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		$3.00 ea.	$3.00 ea	$4.25 ea.	
#'s					
				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

THE COMPUTER CORNER

A Column by Bill Kibler

Well, the last month has been very productive, and I have many things to report. The first topic is the great find I made, a Sage/Stride computer for $200. Pretty low price and at first I wasn't sure about it. The ad said that it didn't work, and it was for parts only. Well, let me say, that if it was for parts, all parts machines should come this way.

The story of this poor user goes like this—he bought the Sage five years ago and used it a lot. A few months ago, it died, and he took it to a dealer. After trying to fix it, the dealer gave up and told him he would have to buy a new board, and that would cost over a $1000. The owner had seen the MACs and so bought one of those, and gave up on the Sage. What I got was a broken machine with all the manuals, schematics, and software, not to mention a broken terminal for an extra $100. I was going to talk him down a little, until I saw what he was parting with—everything. Buys like this are usually short on manuals or software, but this person kept everything—in mint shape too.

68K Trouble-Shooting

When I opened the thing up and started checking what the dealer did against the schematic, it became obvious the dealer did not know anything about 68000s. Having fought with them before, I knew some of the pitfalls possible in this device. The 68K is not like most Intel CPUs, at least in how it does some functions. If you plan on working on the 68K, a Motorola factory information manual is needed, because some of the features are much different than Intel's. The chips removed from the Sage had to do with the HALT line. Typically in other CPUs this line is pulled low when memory or some I/O needs to have the CPU wait while it get things ready. The Sage use of the line is in fact that way, however the 68K can do other things with this line.

When Motorola designed this chip, some checks and options were added. They felt that if something was wrong with the system response, why continue operation? The answer was making the HALT a bi-directional line. This means if something is not going right, like incorrect response from memory, shut down or make HALT active. The RESET line works the same, you can issue a command and force all your hardware to reset from within a program. Heaven help the programmer who accidentally puts a RESET command out and then doesn't reinitialize his hardware, they will wonder why the machine no longer talks to them.

Several chips on the Sage which feed the HALT line had been replaced, and yet the HALT was still active. These chips had been soldered in, and also a solder bridge had been created. If one of those chips had been the problem, the poor replacement of it didn't help. Let me say that I feel that whenever you replace a soldered in device you should replace it with a socket, but they did not. The use of sockets assists in trouble shooting and replacement, although for rough use, sockets are not recommended. Most desktop service can get by with all sockets, but sockets cost more, and chips will work loose over time, so therefore they are not usually installed.

When I picked up the unit from this gentleman, I opened it to just be sure it had not been badly abused by the dealer. At that time I noticed a delay line on the board and had remarked then that it mostly like would be that chip. After fixing the solder bridge and checking that nothing was wrong with the HALT line, I checked the delay line and found it dead. This delay line feeds the memory two different ways and would account for the HALT. The 68K reads the first two memory positions from ROM, pushes and sets PC registers and then jumps to the PC. Without good memory, those pushes don't work and thus a HALT is issued. I didn't have an exact replacement delay line around to try, but I did have one close enough that when I put it in, the machine fired enough to tell me it had BAD memory before shutting down. Before this test, it would not do anything at all.

I called the manufacturer in Reno and they sent me the $6 delay line for $17. Actually I got three, as the line is a special one with two separate delays. They knew quickly which one, as it is the most common failure item. This prompts me to review what usually fails in computers. My list of failure items agrees with Murphy's law and as such there is little you can do to prepare for the failures. Delay lines are high on my list of failure items. They can be very hard to get, although more places carry standard units now than before. The only item higher on my list, is PALs, as these units are available only from the original manufacturer. Should the maker of your machine go out of business you are lost completely, as they seldom publish or make available even the algorithms for them. Another common failure item is the 82S series of PROMS, these small sized items were used for address decoding until PALS came along. While at TELETEK, we published a table of the PROM coding and it was possible to both reprogram one and trouble-shoot it based on the table. PALS provide very little information to help in trouble-shooting. Normally I just check for output activity, keeping in mind that some of the relationships might never be reached to activate that line.

All these items have a tendency to run hot, which probably accounts for their failures. The only 74 series of chips that can fail often are the line drivers. These devices feed the buses and as such will have more than one load, quite often running at their design limits. 74LS374/3, 74LS367/8T97 are common units that fail often due to overuse. Other than those, I haven't found any special devices or situations, other than abuse or poor designs. If the power supply voltages don't get to far astray (up or down) most devices will last forever it seems.

I got the Sage up and running and have ported over the KOS-ONE operating system, which I cover elsewhere. What all this has done however is take some time away from my newest toy, the NOVIX.

NOVIX

As most of you know I have become a user of FORTH as a language for both speed and ease of programming. Well, Chuck Moore the inventor if you will of FORTH, helped design a gate array chip

(Continued on page 11)