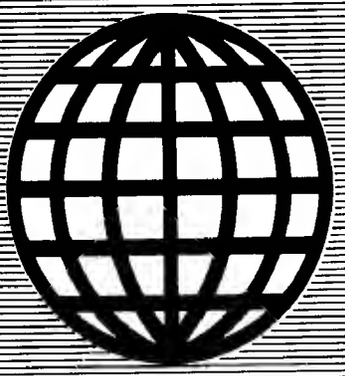


The COMPUTER JOURNAL

Programming - User Support
Applications

Issue Number 54 January / February 1992 US$3.95

Z-System Corner
Ten Years of ZCPR

B. Y. O. Assembler

Local Area Networks

Advanced CP/M

ZCPR On a 16-Bit Intel Platform

Real Computing

Interrupts and the Z80

8 Mhz on an Ampro

Hardware Heaven

What Zilog Never Told You About the Super8

An Arbitrary Waveform Generator

The Development of TDOS

The Computer Corner

ISSN # 0748-9331

Now $4.95 Stops The Clock On Over 100 GEnie Services

For the first time ever, enjoy unlimited non-prime time* usage of many popular GEnie℠ Service features. For just $4.95 a month. Choose from over 100 valuable services including everything from electronic mail and stock closings to exciting games and bulletin boards. Nobody else gives you so much for so little.

You can also enjoy access to a wide variety of features like software libraries, computer bulletin boards, multi-player games, Newsbytes, and the Computer Assisted Learning Center (CALC) for just $6.00 per non-prime hour for all baud rates including 2400. That's less than half of what some other services charge. Plus with GEnie there's no sign-up fee.

Now GEnie not only gives you the information and fun you're looking for. But the time to enjoy them, too.

Follow these simple steps.

1. Set your modem for half duplex (local echo), at 300, 1200 or 2400 baud.
2. Dial toll free 1-800-638-8369. Upon connection, enter HHH.
3. At the U#=prompt, enter XTX99486,GENIE then press RETURN
4. Have a major credit card or your checking account number ready.

For more information in the U.S. or Canada, call us voice at 1-800-638-9636.

TCJ readers are invited to join us in the CP/M SIG on page 685 and the Forth Interest Group SIG on page 710. Meet the authors and editors of *The Computer Journal!* Enter "M 710" to join the FIG group and "M 685" to join the CP/M and Z-System group.

We'll meet you there!

JUST $4.95

Moneyback Guarantee

Sign up now. If you're not satisfied after using GEnie for one month we'll refund your $4.95.

*Applies only in U.S. Mon.-Fri., 6PM-8AM local time and all day Sat., Sun., and select holidays. Prime time hourly rates $18 up to 2400 baud. Some features subject to surcharge and may not be available outside U.S. Prices and products listed as of Oct.1, 1990 subject to change. Telecommunications surcharges may apply. Guarantee limited to one per customer and applies only to first month of use. GE Information Services, GEnie, 401 N. Washington Street, Rockville, MD 20850. © 1991 General Electric Company.

The Computer Journal

Founder
Art Carlson

Editor/Publisher
Chris McEwen

Technical Consultant
William P. Woodall

Contributing Editors
Bill Kibler
Matt Mercaldo
Tim McDonough
Frank Sergeant
Brad Rodriguez
Clem Pepper
Richard Rodman
Jay Sage

The Computer Journal is published six times a year by Socrates Press, P.O. Box 12, S. Plainfield, NJ 07080. (908) 755-6186

Subscription rates• Within US: $18 one year (6 issues), $32 two years (12 issues). Foreign (surface rate): $24 one year, $44 two years. Foreign (airmail): $38 one year, $72 two years. All funds must be in U.S. dollars drawn on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: *The Computer Journal*, P.O. Box 12, S. Plainfield, NJ 07080, telephone (908) 755–6186.

Registered Trademarks

TCJ The Computer Journal

Issue Number 54 January / February 1992

Editor's Desk

By Chris McEwen

Ten Years of ZCPR

The lead article this issue commemorates the tenth anniversary of the release of ZCPR to the public domain. This event is more significant than it may seem. ZCPR, the first building block of what we today call Z-System, represents a monumental contribution to public domain programming. To this day, I know of no other project more ambitious.

ZCPR is an essential element of an operating system. Few operating systems have been released in this way. It was developed to improve and replace the product of a major corporation. It is always used by licensees of the commercial version who use it in preference. In comparison, Minix was developed to provide an educational sample, a limited model of a more powerful system for those who lacked the full system.

ZCPR stands as a monument to cooperation and teamwork. It was first released through the Amateur Computer Group of New Jersey (ACGNJ), one of the world's largest and most prestigious user groups. Work today continues through the joint efforts of people who gather electronically on Z-Nodes and on computer networks around the world. While this community has its elders as any would, it remains a grassroots movement. It has never been the domain of lone wolves.

Work on ZCPR continues after a decade. This is an eternity in the world of micro computers! To put this is perspective, the Altair was introduced in 1976. If we consider that event the beginning of recorded history for this industry, then 1982 would equate to the time of Moses! Yet ZCPR continues as a viable, useful system.

I asked Jay Sage if he would do the honors for us. We know Jay as the author of ZCPR versions 3.3 and 3.4, and the *TCJ* series on ZCPR that has continued uninterrupted since 1987. He has been a mover and shaker in this community nearly from the beginning. His bulletin board system is Z-Node #3, from a universe that at its height was over 70.

Join us in this celebration of the human spirit. May our next ten years be as good as our first!

Bridger's Back.

Bridger Mitchell returns with his *Advanced CP/M* column this issue! This is big news!

Bridger joined forces with Derek McKay to form Plu*Perfect Systems in 1985. Their first product was an upgrade to Perfect Writer. They wrote the Kaypro TurboROM, which became a Must Have upgrade for Kaypro owners and was the beginning of Bridger's work in fixing hurting operating systems.

The two next wrote Backgrounder which allowed users to

See Editor, page 34

Reader-to-Reader

In issue 53, a reader from Bonn, Germany refers to a rumored way to add a SCSI controller to a Z80 system by unplugging the CPU and installing an adapter card. This is probably a reference to a product sold by Emerald Microware somewhere in Oregon; they used to advertise in *Micro Cornucopia*, but I don't seem to have any issues handy.

The product that they sell is a connection to the WD1002-05 ST506 hard disk controller. Although this controller is generally believed to be either SASI or SCSI, this is not the case; it is just a simple 8-bit interface involving 3 address lines, 8 data lines, a Chip Select, Read, and Write. The module in question decodes the address for the WD1002-05 and places the stuff needed by that controller onto a connector that can be cabled over to the WD1002-05. In addition to this module, Emerald Microware also sells the WD1002-05, software to control it, spare parts for Kaypros, and lots of other nifty stuff.

I have purchased this module and a WD1002-05, but their NorthStar format and mine disagree on the interleave of the disk so I have not had any experience with their software. I intended to write my own BIOS for the thing anyway, so it's no biggie. Just haven't gotten around to it. I have done enough to know that the module is correctly communicating with the WD1002-05, however.

By the way, no one at Western Digital remembers the WD1002-05, so don't call them and ask about it. It also

See Reader, page 44

Letters to the editor and other readers are welcome. Submit to The Computer Journal, Post Office Box 12, South Plainfield, NJ 07080-0012. Letters may also be electronically submitted via Internet to "cmcewen@gnat.rent.com," via GEnie™ to "TCJ$" or to Socrates Z-Node at (908) 754-9067. Submission implies permission to publish your letter unless otherwise stated. Letters may be edited as necessary.

Z-System Corner

Ten Years of ZCPR

By Jay Sage

On February 2, 1992, exactly ten years have passed since the first version of ZCPR was released. I have been involved with it one way or another most of that time, and I think it is amazing how vibrant the activity in the field still is.

Our editor, Chris McEwen, had hoped that we could make this issue a special celebration of ZCPR with contributions from some of the original developers, most notably Richard Conn. I exchanged email with Richard several times about this, but he never picked up on it. Since of all those still active in the Z community I may be the one who goes back the farthest, it is perhaps fitting that I take on the task.

Announcement

Before I start on that, I do have one important announcement to make. I would like to call your attention to the new *Sage Microsystems East* ad in this issue. You will notice that there have been quite a number of significant price reductions. We hope that by lowering the entry price to Z-System from $70 to $49 we will encourage more people who still do not use Z-System to try it out.

The History of ZCPR

Much of the material in this column comes from the introductory chapter of my book, the *ZCPR33 User Guide*. When Echelon made my revision of ZCPR3 the official product release in 1987, naturally they wanted a manual to go with it. Besides including all the necessary technical information, such as what the new command processor did and how it should be installed, I also included two other items that were of great importance to me: a statement of what I was trying to achieve with ZCPR33 and the history that led up to it. Here in *TCJ* I often talk about the goals of Z-System; this time I will review some of the history.

ZCPR1

"Don't you know about ZCPR!" I remember very well being greeted with that exclamation from one of the veteran club members when, as a neophyte computer user, I attended a CP/M computer club meeting. He could not believe that someone would still be using standard CP/M. I soon felt the same way and still do today!

The ZCPR he was referring to was what we would now call ZCPR1. ZCPR, which stood for "Z80 Command Processor Replacement," was the work of a group of computer hobbyists who called themselves "The CCP Group." They were Frank Wancho, Keith Petersen, Ron Fowler, Charlie Strom, Bob Mathias, and Richard Conn. Richard, as we will see, was the main force behind the effort.

Ron Fowler is well known as the author of the MEX telecommunications program, which I still use and enjoy immensely. Keith Petersen wrote a simplified version of Ward Christensen's CBBS, the original computerized bulletin board system. Keith's program was called MINICBBS; my own customized version of it runs to this day on my Z-Node. It is, to be sure, outmoded, but it gives me a sense of connection with history that I still treasure.

Keith Petersen was for a long time a sysop of one of the finest BBS systems in the country, Royal Oak. Though it branched out to MS-DOS software, it never neglected CP/M. It was, perhaps, unique in that callers to Royal Oak found themselves immediately at the operating system prompt. There was no request for a name or password. If you wanted to use MINICBBS, you had to invoke it yourself.

Frank Wancho is involved in the administration of the SIMTEL20 computer at the White Sands Missile Range. This machine houses a huge archive of CP/M programs (and many others). Keith is on contract to SIMTEL20 to help maintain the collections. I continue to see both their names. Frank, via email, gave me some of the information on the birth of ZCPR.

Sometime around 1981 Richard Conn sparked the group's enthusiasm

---

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR command processor, his ARUNZ alias processor and ZFILER, a "point-and-shoot" shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (MABOS on PC Pursuit, 8796 on Starlink, pw=DDT). He can also be reached by voice at 617-965-3552 (between 11 p.m. and midnight is a good time to find him at home) or by mail at 1435 Centre Street, Newton Centre, MA 02159. Jay is now the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail, or chatted with live at the Wednesday real-time conferences (10 p.m. Eastern time).

In real life, Jay is a physicist at MIT, where is tries to invent devices and circuits that use analog computation to solve problems in signal, image and information processing. His recent interests include artificial neural networks and superconducting electronics. He can be reached at work via Internet as SAGE@LL.MIT.EDU.

over rewriting the CP/M console command processor, or CCP, to take advantage of the more efficient and elegant opcodes of the new Zilog Z80 microprocessor. The people in the CCP Group were not in physical proximity. I believe that they maintained contact, as we do today, via electronic mail. Frank Wancho provided the computer access that made that contact possible.

The original ZCPR was released by SIG/M of the Amateur Computer Group of New Jersey

With some space opened up in the CCP, the programmers were able to add a number of convenient new features. The most important new concept was that of a search path for COM files. With CP/M version 2, Digital Research had introduced user numbers, but the way they were implemented made them virtually worthless, because there was no way from one user area to run or access files in another user area. ZCPR, with its ability to automatically search drive A/user 0, overcame this problem and opened up the possibility of putting the new user areas to effective use.

Also introduced with ZCPR was the GO command, which permitted the most recently executed transient program to be run again without having to reload it from disk. That was a real a boon in those days of slow floppy drives. Many small—but very useful and helpful—improvements were made in the resident commands. For example, in CP/M, when a REN or SAVE command specified a destination file that already existed, the command would simply abort. The user would then have to erase the old file manually and start over again. With ZCPR, the REN and SAVE commands made life easier by asking the user if the old file should be overwritten.

The original ZCPR was released to the public on a disk published by SIG/M (Special Interest Group / Microcomputers), the public-domain software distribution arm of the Amateur Computer Group of New Jersey (ACGNJ). The disk was volume 54, dated February 2, 1982. Interestingly enough, this is volume 54 of *The Computer Journal* that you are reading ten years later!

Several additional refinements were made to ZCPR by other programmers, leading to a train of development known as NZCPR (New ZCPR). Version 1.6 of NZCPR was released on SIG/M volume 77 at the end of October, 1982. This branch eventually reached version NZCPR21, a version never published in disk form but distributed over the remote access computer system network.

Jim Byram, of the Boston Computer Society CP/M Group, produced a privately distributed version of NZCPR using only Intel 8080 code, which showed that efficient coding, and not simply the use of the new Z80 opcodes, was a major factor in improving the command processor. Jim, by the way, may be the one who made the remark to me that I quoted earlier. I eventually became the leader of that group, which merged with several others and ultimately became the Zi/Tel Group, of which I am now the CP/M director and the bulletin board sysop. That group supports CP/M, Z-System, and MS-DOS.

ZCPR2

While ZCPR1 was a significant improvement over CP/M, it was not a revolutionary advance. Richard Conn, however, had a vision of a truly advanced operating system, and he continued the development. On February 14, 1983, almost exactly one year after ZCPR1 appeared, ZCPR2 was released in a set of ten SIG/M volumes (98-107), an unprecedented and monumental contribution of public-domain software.

ZCPR2 made a very significant conceptual advance: it used memory buffers in protected memory above the BIOS to hold new operating system modules. The command line, which had always resided in the command processor, was put in one of these buffers so that it would not be destroyed by warm boots, during which a fresh copy of the command processor is loaded from disk. In that way multiple commands on a line could be implemented.

The command search path was also placed in one of these buffers instead of hard-coding it into the command processor. In this way the search path could be changed by the user at any time. The concept of named directories was also introduced, using still another memory buffer to store the index of names.

Many of the utilities that we are familiar with in ZCPR3 first appeared with ZCPR2. These include ZEX, WHEEL, HELP, PATH, PWD, MKDIR, and MENU. A rudimentary shell concept was used in MENU. When this program placed a command into the multiple command line buffer, it would always add its own name at the end of the command sequence so that control would eventually return to MENU. This worked fine for single levels of shells. Extended command processing was also introduced with ZCPR2.

The ZCPR2 documentation alone ran to more than half a megabyte

The ZCPR2 documentation, alone, ran to more than half a megabyte! It included a concepts manual, an installation manual, a users guide, and a rationale manual (I guess Rick felt he had to prove he wasn't crazy in doing all this wonderful stuff).

Shortly after the initial ZCPR2 SIG/M release, an upgrade to version 2.3 was published in volume 108. Up to this point ZCPR2 still followed in the tradition of ZCPR1 and used Zilog opcodes. The features of ZCPR2 were now so exciting, however, that owners of computers based on Intel's 8080 and 8085 microprocessors wanted to have them, too. Charlie Strom, a member of the original CCP Group and well-known later as the sysop of the Compuserve CP/M Special Interest Group, converted the command processor code and some of the key utilities to Intel-compatible code and released the result in SIG/M volume 122. At the time, believe it or not, I was using at work an Intel MDS-800 microprocessor development system, the computer for which Gary Kildall, then at Intel, invented CP/M, and I remember very well bringing up this 8080 version of ZCPR2. It was marvelous!

ZCPR3

But ZCPR2 was by no means the end of the evolution. On Bastille day, July 14, 1984, not quite a year and a half after ZCPR2, Richard Conn offered ZCPR version 3 in the form of

another nine volumes of SIG/M disks (184 to 192). At this point more than 10% of all the software ever released by SIG/M had come from one contributor—Richard Conn!

One time when I was talking with Richard, I must have expressed my amazement at the incredible amount of software he had written and released to the public. I was equally impressed by Richard's response. He said that the code that others had offered to the public had taught him and helped him so much that he felt a tremendous obligation to contribute what he could to the community. He certainly did that! And that same spirit still pervades the 8-bit community.

ZCPR3 brought both significant new concepts and major refinements

ZCPR3 brought both significant new concepts and major refinements. Three of the innovations were flow control, error handling, and the message buffer.

Flow control made it possible to achieve a vastly higher degree of automated operation, since the command processor was no longer dependent on the user for all command decisions but could now make wide-ranging decisions on its own. The message buffer made possible communication between the command processor and programs and between successively run programs.

Error handlers made it possible for improperly entered commands to be corrected, an important facility to have in connection with multiple commands on a line. Having to retype a single command after a mistake had been bad enough; having to retype a whole, long string of commands because of a single mistake seriously discouraged one from making use of the multiple command facility.

ZCPR3, by the way, unlike it predecessors, was written so that it could be assembled to either Intel or Zilog opcodes. In the former case, the code was considerably longer and fewer features could be included, but it would work on an 8080 or 8085 computer.

ZCPR31

The chain of refinements to ZCPR3 that led to version 3.3 started in March, 1985, when I produced a private, experimental version of ZCPR3 called ZCPR31 for use on my Z-Node. It was modified so that the command processor would get the values for maximum drive and user from the environment descriptor (more on this later).

This was my first close look at operating system code, something that had always frightened me, as I am sure it has many others. There is a mystique about those words, "operating system," that makes one think that only the most advanced programmers could possibly understand the code. In fact, I discovered that the code did not look much different from that in ordinary utility programs. To my amazement, I was able to make changes that worked and improved the CCP. The most significant advances occurred in August, 1985, when three further major enhancements were introduced.

First, the code was changed to prevent the infinite loop that Z30 experienced when the specified error handler could not be found (perhaps because the path was changed or the error handler renamed). In that situation, a command error would invoke the error handler. When the error handler could not be found, that constituted another error that caused the error handler to be invoked, and so on until one pressed the reset button or turned off the power.

Second, the code was modified so that it could determine the addresses of the RCP, FCP, and NDR modules from the environment and respond to dynamic changes in these addresses.

Finally, additions were made to the code that allowed an extended command processor to return control to the command processor if it also could not resolve the command. The command processor would then invoke the error handler. Now the extended command processor really was a full-fledged extension of the CCP, and a ZCPR3 system could take advantage of both extended command processing and error handling. The same mechanism also made it possible for ordinary programs to initiate error handling.

In January, 1986, the first steps were taken to fix serious bugs in the way the minimum path and root path were computed. The fix, however, had errors of its own, and it was not until June, 1986, that Howard Goldstein finally implemented a complete and proper solution.

The next major set of advances came in March, 1986, when Al Hawley, sysop of Z-Node #2 and now a familiar *TCJ* author, introduced several new concepts. One was a new way to implement wheel-protected commands (commands that can be executed only by specially authorized users). In Z30 wheel protection had to be hard coded into the command processor (and RCP), and when one of the restricted commands was invoked with the wheel off, an error message resulted. Al introduced the idea of setting the high bit of the first character of a command to signal that the command was off-limits to non-wheel users.

There is a mystique about those words, "operating system," that makes one think that only the most advanced programmers could possibly understand the code

This concept had several important advantages. First, the code was shorter. Second, the new code automatically made the same technique apply to commands in other modules (RCP and FCP), so that wheel-checking code could be eliminated from those modules. Third, when the wheel byte was off, wheel-protected commands instead of displaying an error message simply vanished as far as the command processor was concerned. In this way, transient programs or aliases with the same name as the resident command could automatically step in and provide whatever action the system implementer desired.

Al Hawley also introduced two concepts that made dealing with secure systems easier. He made it possible for the command processor to determine dynamically whether or not to recognize the DU form of directory reference in response to the setting of the DUOK flag in the environment, and he allowed the option of bypassing password checking when the wheel byte was set. These features made it possible for a sysop or system implementer to live comfortably with a secure system (though they did not make life any easier for the restricted user).

The last major advance that occurred in the development

of ZCPR31 resulted from a conversation I had with Bruce Morgen in July, 1986. We were discussing the annoying way that ZEX functioned under shells, with the shell program being reloaded for each command line, only to realize that ZEX was running. It would then feed the next command line from ZEX to the multiple command line buffer. I conceived a small change in the code that made this problem vanish in a flash.

ZCPR33

At the very end of January, 1987, I got a call from Echelon. Richard Conn had decided to discontinue his involvement with ZCPR3, and Echelon asked if I would be willing to write the official ZCPR version 3.3 release based on the experimental ZCPR31. I agreed. During the months of February, March, and April of 1987 an enormous amount of additional development took place, the results of which are described in detail in the *ZCPR33 User Guide.* Only some key concepts will be mentioned here.

The decision was made no longer to make any attempt to support 8080/8085 computers. The code was written using Zilog mnemonics, and extensive use was made of Z80-specific instructions, including relative jumps, block moves, block searches, direct word transfers to register pairs other than HL, 16-bit subtractions, and the alternate register set. This approach has continued to the current ZCPR34. To my knowledge, no one has even tried to make an 8080 version; there just are not many of those machines still in operation.

One of the nicest features introduced with ZCPR33 was the automatic installation of programs. Until this point, before a ZCPR-aware program could be used, it had to be "installed" for the specific system configuration. If one forgot to do this, the program would likely behave in bizarre ways, and this was a very common source of difficulty for new and experienced users alike.

In ZCPR2 installation was a very elaborate procedure in which a large block of code had to be patched using the special GENINS utility. With ZCPR3 the information about the system configuration was placed in a memory buffer (called the environment or ENV) where all programs could access it. More importantly, the system configuration could be changed without reinstalling all the programs. Now installation amounted only to patching the ENV address into the program.

As soon as I heard that Richard Conn had figured out a way to eliminate this annoying installation step, the solution became obvious to me as well. Since the command processor already loads a program from disk, and since it already knows the ENV address, why couldn't it install the address directly into the memory image of the program? That's just what it does.

One truly revolutionary concept was introduced with ZCPR33. Until that time, all CP/M transient programs were loaded to and ran at a standard address, 100H. With CP/M there was no reason to do otherwise, but with ZCPR3 there was. From the time of ZCPR1, I had become quite accustomed to using the GO command to rerun the previous program. To my puzzlement, GO sometimes produced bizarre results under ZCPR3.

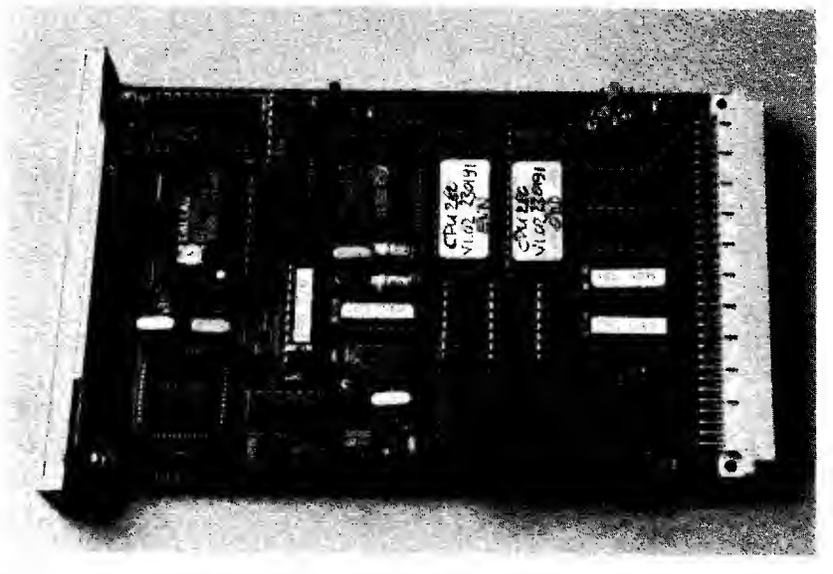
Under CP/M, programs get loaded only when the user instructs the system to run them. Under ZCPR3, however, there are quite a few programs that are loaded and executed automatically by the command processor. These include extended command processors, error handlers, shells, and transient (COM) versions of otherwise resident commands, such as ERA or REN. Sometimes, using the GO command resulted in rerunning these programs instead of the last program the user specified.

One day as I was working on the ZCPR33 code, I noticed that a trivial change would allow the command processor to load a file to an address other than 100H. This, I realized, could overcome the problems with the GO command. User programs could be loaded, as usual, to 100H, but programs invoked automatically by the command processor could be loaded to a higher address, such as 8000H. User programs in low memory would not be overwritten, and the GO command would still be able to rerun them.

One more group of major innovations was introduced with ZCPR33. ZCPR30 provided a number of security features that made it particularly suitable for use on a remote access system (BBS). The so-called wheel byte could be used to control access to both resident and transient programs. Dynamically changeable limits on the range of drives and user numbers and named directories with passwords could keep callers out of certain disk areas.

This security made it possible to allow remote users to run a system directly from the command line prompt, in sharp contrast to MS-DOS remote systems, where a user who gets to the command prompt has free reign to access or destroy any part of the system.

The security system under ZCPR30, while fully effective, however, could be an unnecessary nuisance. For example, there could be situations where a user could access a directory area by name, because it had no password, but not by drive/user value, because they exceeded the allowed range. Under ZCPR33, if a directory is accessible by name, then it could also be accessed by drive/user.



The CPU280, designed and built by Tilmann Reh, was described in issue 53. It uses a Zilog Z280 at 12.5 MHz and accepts up to 4 MB of RAM.

Tilmann Reh, Jay Sage and Uwe Herczeg with a CPU280 during Jay's visit to Germany in June 1991. Uwe is working on software to integrate Tilmann's IDE controller to this computer. The group met at Uwe's computer store in Brackenheim.

ZCPR34

The current state of the art of the ZCPR command processor is version 3.4. It was first released some time around March of 1988 along with NZCOM and Z3PLUS and was described in issue 32 of *TCJ*. Relative to Z33 it was an evolutionary advance, a refinement; there were no radical new ideas, as there had been in Z33. Nevertheless, the changes were significant and useful. There have been several minor revisions since the original release.

One change introduced with ZCPR34 was an extended environment descriptor. We removed some information that had proved to be of little use and added new information. The most important addition was a drive vector word. The ENV always had a max-drive byte that specified the highest letter drive available on the system. However, this was not adequate for systems that had drives that were not contiguous, such a A:, B:, and E:. The new drive vector tells exactly which drives are available for use at any time.

The new ENV also contains the addresses of the CCP, BDOS, and BIOS and the sizes of the first two. This is to prepare us for some future enhancements in which we will not necessarily adhere to the standard component sizes that were specified in the original CP/M. Hal Bower and Cam Cotrill, as part of the development of a new banked version of ZSDOS (which is nearing release), have been experimenting with a CCP that is larger than the usual 2K and a banked DOS that will be significantly smaller than the standard 3.5K.

I mentioned earlier that ZCPR33 had rationalized the implementation of directory security so that any area accessible by name would also be accessible by drive/user, even if the drive or user exceeded the limit set in the environment. With ZCPR34 the symmetry was completed. Now if there is a password-protected directory that could be accessed freely using the DU: format, then its password will be ignored. Now there will never be directory areas that can be accessed in one way but not the other.

The extended command processor interface was liberalized so that commands that would formerly have been considered illegal and processed immediately as errors, such as those with wildcard characters ('?' or '*') or with an explicit file type, can be passed to the ECP. For example, the ALIAS.CMD file that defines aliases for the ARUNZ extended command processor can now have an entry for the command '?' that invokes the program HELP.

The most significant advance in ZCPR34 was support for what we now call a type-4 program. Type-3 programs, as we described earlier, are loaded and run at an address other than 100H, but the address is still fixed at the time the program is compiled. It was clear to me at the time I wrote Z33 that it would be ideal if the load/run address of a program could be determined dynamically (that is, at the time it is loaded by the CCP). However, I opted for the very simple code that sufficed for handling the type-3 program.

Joe Wright was not satisfied with that compromise and soon wrote an initial implementation of a type-4 program, which would relocate the code automatically to the top of free memory. With a lot of cooperation between us, we honed the approach to the point where it functioned very nicely and added very little code to the command processor.

The secret to this lay in Joe's use of what is called a PRL (page relocatable) program for the executable file. The details of this are described in *TCJ* issue 32. The standard PRL file begins with two 128-byte header records, and I suggested placing the code required to calculate the proper load address and the code to perform the address relocation in these header records rather than in the command processor itself. Joe found a brilliant way to implement this.

Not only did this approach keep the CCP code shorter, it also made the whole type-4 program more flexible by making it independent of the command processor. My next *TCJ* column will introduce the first examples of alternative versions of the type-4 loader routines that are placed in the PRL header. These new headers can be installed by the user in any existing type-4 program to change the way the program relocates itself. This is another example of the beauty of the modular approach that has been one of the hallmarks of ZCPR.

My next column will also introduce the latest revision of ZCPR34, version 3.4E. Howard Goldstein prepared this version by integrating a number of ideas, most notably a small change in the type-4 loader code to make it even more flexible than it was originally.

So, after ten years, an eternity in the computer industry, ZCPR—the concept Richard Conn initiated—is still developing and still challenging the creativity of users and programmers alike. As always, these developments arise from the cooperation of a large community of people willing and eager to share ideas.●

One of the advantages of being disorderly is that one is constantly making exciting discoveries.

—Alan Alexander Milne

B. Y. O. Assembler

A 6809 Forth Assembler

By Brad Rodriguez

Introduction

Part 1 of this two-part series, published in issue 52, described the fundamental concepts used when writing postfix, structured assemblers in Forth. This article examines, in detail, an actual assembler for the Motorola 6809 microprocessor.

I will begin with the "programmer's guide" for the 6809 assembler. I've found, when building assemblers, that it's helpful to write this first. This is good practice, and is also in keeping with *Rodriguez' First Rule for Metacompiler Writers:* always keep in mind what you want the result to look like!

In the off-chance that someone actually wishes to use this assembler on a 6809 project, I've also included a conversion chart from Motorola to Forth notation. If you're ever writing an assembler for others to use, please do them a favor and make one of these charts!

Finally, I'll go over the source code, and (hopefully) explain all of the obscure tricks and techniques.

The complete source code for the 6809 assembler accompanies this article. It is written for a fig-Forth derivative, and so will require translation for 79- or 83-Standard machines. This code was originally a Forth screen file; it has been translated to a text file for the editor's convenience.

Programmer's Guide

The syntax of 6809 assembler instructions is:

```
operand  addressing-mode  opcode
```

For many instructions the addressing mode is optional. Some instructions (TFR and EXG) have two operands.

Valid non-indexed addressing modes are:

```
nn #          immediate value
nn <>         direct (DP-page) addressing
nnnn          extended addressing
nnnn []       extended indirect addressing
```

Valid indexed addressing modes are:

```
r 0,          zero offset
r nn ,        constant offset
r A,          accumulator A offset
r B,          accumulator B offset
r D,          accumulator D offset
r ,+          autoincrement by 1
r ,++         autoincrement by 2
r ,-          autodecrement by 1
r ,--        autodecrement by 2
nn ,PCR       constant offset from PC
```

"r" is one of the registers X, Y, U, or S. nn is a twos-complement (signed) offset.

All of the indexed modes except autoincrement/decrement by 1 also have an Indirect form. This is specified by appending the suffix [] after the addressing mode, e.g.:

```
r ,++ []      indirect autoincrement by 2
```

The TFR and EXG instructions take the form:

```
src dst TFR          src dst EXG
```

where src and dst can be any of the 16-bit registers D, X, Y, U, S, PC; or any of the 8-bit registers A, B, CCR, DPR.

Branch offsets for relative jumps are computed internally by the assembler; the operand for a relative jump is the destination address.

The following control structures are provided by the assembler:

```
cc IF,   ...   THEN,
do code if cc satisfied

cc IF,   ..1..   ELSE,   ..2..   THEN,
do code ..1.. if cc satisfied, else do code ..2..

BEGIN,   ...   cc UNTIL,
loop through code until cc satisfied;
always executes at least once

BEGIN, ..1.. cc WHILE, ..2.. REPEAT,
loop through code until cc satisfied;
exit is evaluated and taken after
code ..1.. is executed (always at
least once)
```

where the condition code cc is any of the following:

```
CC carry clear       HI higher
CS carry set         HS higher or same
EQ equal/zero        LE less than or
GE greater or equal     equal
```

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

```
\ ------------------------------------------------------
\ 6809 assembler: utilities              (c) 14 11 85 BJR
\
VOCABULARY ASSEMBLER IMMEDIATE  ASSEMBLER DEFINITIONS  HEX
: WITHIN   ROT SWAP OVER   \ n lo hi — f | test within limits
   < ROT ROT > OR 0= ;
: 8BIT?   -80  7F  WITHIN ;
: 5BIT?   -10 0F WITHIN ;
: (,)  , ;
\ : ALIGN ;  ( for byte-addressing hosts)

\ Buffoonery to allow byte-oriented assembly on word machines.
\ Assembler does ALIGNing, thus always knows when to de-ALIGN!
\ Include if host is word-aligned.
0 VARIABLE ALIGNED          \ flag indicating HERE was adjusted
: ALIGN    HERE 1 AND DUP ALLOT ALIGNED ! ;      \ align HERE
: DEALIGN  ALIGNED @ MINUS ALLOT  0 ALIGNED ! ; \ cancel ALIGN
: HERE     HERE ALIGNED @ - ;  \ true HERE for address calc'n
: C,   DEALIGN C, ALIGN ;        \ redefine C, for assembler

\ ------------------------------------------------------
\   6809 assembler: addressing modes     (c) 03 06 85 BJR
\
: W,   DUP >< C, C, ;        \ store word as hibyte, lobyte
: OPCODE,                    \ store opcode with prefix (if any)
   DUP FF00 AND  IF W, ELSE C, THEN  ;

30 VARIABLE MODE   \ 0=immed,10=direct,20=indexed,30=extended
: #   0 MODE ! ;
: <>   10 MODE ! ;

: INDEXREG   20 MODE !   \ rval postbyte — postbyte |
   SWAP  1-  DUP 0 3 WITHIN 0= 3 ?ERROR  \ must be x,y,u, or s
   20 * OR ;                          \ put reg # in postbyte

: XMODE   <BUILDS (,)   \ postbyte —  | Simple Indexed Modes
   DOES>  @ INDEXREG ;  \ rval — postbyte

84 XMODE 0,    86 XMODE A,    85 XMODE B,    8B XMODE D,
80 XMODE ,+    81 XMODE ,++   82 XMODE -,    83 XMODE —,

\ ------------------------------------------------------
\   6809 assembler: addressing modes     (c) 04 06 85 BJR
\
: ,      SWAP 89 INDEXREG ;    \ rval n — n postbyte |
: ,PCR   20 MODE !  8D ;       \ n — n postbyte |

: []   MODE @ 20 =     \ Indexed:  postbyte — postbyte
                       \ Extended:         n — n postbyte
   IF  DUP 9D AND 80 = 3 ?ERROR  10 +     \ Indexed Indirect
   ELSE  20 MODE !  9F THEN ;             \ Extended Indirect

: RESET   30 MODE ! ;

\ register definitions
0 CONSTANT D     1 CONSTANT X     2 CONSTANT Y     3 CONSTANT U
4 CONSTANT S     5 CONSTANT PC    8 CONSTANT A     9 CONSTANT B
0A CONSTANT CCR     0B CONSTANT DPR

Y CONSTANT IP    U CONSTANT SP    S CONSTANT RP    X CONSTANT W

\ ------------------------------------------------------
\   6809 assembler: inherent instruction(c) 03 06 85 BJR
\
: INHOP   <BUILDS (,)       \ opcode — | Inherent Addressing
   DOES>   @ OPCODE, RESET ; \ — | lay one or two bytes

3A INHOP ABX,   48 INHOP ASLA,  58 INHOP ASLB,  47 INHOP ASRA,
57 INHOP ASRB,  4F INHOP CLRA,  5F INHOP CLRB,  43 INHOP COMA,
53 INHOP COMB,  19 INHOP DAA,   4A INHOP DECA,  5A INHOP DECB,
4C INHOP INCA,  5C INHOP INCB,  48 INHOP LSLA,  58 INHOP LSLB,
44 INHOP LSRA,  54 INHOP LSRB,  3D INHOP MUL,   40 INHOP NEGA,
50 INHOP NEGB,  12 INHOP NOP,   49 INHOP ROLA,  59 INHOP ROLB,
46 INHOP RORA,  56 INHOP RORB,  3B INHOP RTI,   39 INHOP RTS,
1D INHOP SEX,  3F INHOP SWI, 103F INHOP SWI2, 113F INHOP SWI3,
13 INHOP SYNC,  4D INHOP TSTA,  5D INHOP TSTB,

\ ------------------------------------------------------
\   6809 assembler: immediate instructio(c) 03 06 85 BJR
\
: IMMOP   <BUILDS (,)      \ opcode — | Immediate Only (8-bit)
   DOES>   MODE @ 3 ?ERROR  @ C, C, RESET ;  \ operand —

3C IMMOP CWAI,  34 IMMOP PSHS,  36 IMMOP PSHU,  35 IMMOP PULS,
37 IMMOP PULU,  1C IMMOP ANDCC, 1A IMMOP ORCC,

: RROP   <BUILDS (,)       \ opcode — | Register-Register
   DOES>   @ C,  SWAP 10 * + C, RESET ;  \ srcrval dstrval —

1E RROP EXG,    1F RROP TFR,

\ ------------------------------------------------------
\   6809 assembler: +mode                (c) 03 06 85 BJR
\
: +MODE    \ operand — operand | modify operand per mode
   MODE @ +  DUP 0F0 AND 50 = IF 0F AND THEN ; \ chng 5x to 0x

\ ------------------------------------------------------
\   6809 assembler: pcrel, cofset        (c) 29 03 85 BJR
\
: PCREL   \ operand postbyte — | lay PC relative
   SWAP  HERE 2+ -  DUP 8BIT?  \ try 8 bit relative offset
   IF  SWAP 0FE AND C, C,       \ it fits...lay postbyte,offset
   ELSE  1- SWAP C, W, THEN ;  \ no good...use 16 bit relative

: NOTINDIR?  10 AND 0= ;  \ postbyte — f | test for indirect

: COFSET   \ operand postbyte — | lay constant offset
   OVER 0= IF  0F0 AND 4 OR C, DROP      \ no offset
   ELSE OVER 5BIT? OVER NOTINDIR? AND IF
      60 AND SWAP 1F AND OR C,           \ 5 bit offset
   ELSE OVER 8BIT? IF  0FE AND C, C,     \ 8 bit offset
   ELSE  C, W,  THEN THEN THEN ;         \ 16 bit offset

\ ------------------------------------------------------
\   6809 assembler: indexed, immed       (c) 03 06 85 BJR
\
: EXTIND    \ operand postbyte — | lay extended indirect
   C, W, ;       \ lay postbyte and operand

: INDEXED   \ operand? postbyte — | lay indexed postatuff
   DUP 8F AND CASE       \ check postbyte for modes w/ operands
      89 OF COFSET   ENDOF       \ const.offset
      8D OF PCREL    ENDOF       \ PC relative
      8F OF EXTIND   ENDOF       \ extended indir
       SWAP C,  ENDCASE ;  \ simple modes, postbyte only

: IMMED   \ operand opcode-pfa — | lay immediate postatuff
   2+ @  DUP 0= 3 ?ERROR     \ test immedsize
   1- IF W, ELSE C, THEN ;  \ lay immed. operand in reqd.size

\ ------------------------------------------------------
\   6809 assembler: general addr instr  (c) 03 06 85 BJR
\
: GENOP   <BUILDS (,) (,)   \ immedsize opcode — | Gen'l Addr
   DOES>   DUP @ +MODE OPCODE,   \ [see below] | lay opcode
      MODE @ CASE 0 OF  IMMED         ENDOF   \ immediate
                  10 OF  DROP C,      ENDOF   \ direct
                  20 OF  DROP INDEXED ENDOF   \ indexed
                  30 OF  DROP W,      ENDOF   \ extended
      ENDCASE RESET ;

: INXOP   <BUILDS (,)       \ opcode — | Indexed Only
   DOES>   MODE @ 20 - 3 ?ERROR  @ OPCODE, INDEXED RESET ;

\ Stack action of general addressing instructions
\ (1) immediate, direct, extended:                operand —
\ (2) all indexed except (3):                    postbyte —
\ (3) const.offset, PCR, extended indir:  operand postbyte —

\ ------------------------------------------------------
\   6809 assembler: general addr instr  (c) 29 03 85 BJR
\
1   89 GENOP ADCA,   1   C9 GENOP ADCB,   1   8B GENOP ADDA,
1   CB GENOP ADDB,   2   C3 GENOP ADDD,   1   84 GENOP ANDA,
1   C4 GENOP ANDB,   1   85 GENOP BITA,   1   C5 GENOP BITB,
0   48 GENOP ASL,    0   47 GENOP ASR,    0   4F GENOP CLR,
1   81 GENOP CMPA,   1   C1 GENOP CMPB,   2 1083 GENOP CMPD,
2 118C GENOP CMPS,   2 1183 GENOP CMPU,   2   8C GENOP CMPX,
2 108C GENOP CMPY,   1   88 GENOP EORA,   1   C8 GENOP EORB,
0   43 GENOP COM,    0   4A GENOP DEC,    0   4C GENOP INC,
1   86 GENOP LDA,    1   C6 GENOP LDB,    2   CC GENOP LDD,
```

```
2 10CE GENOP LDS,     2   CE GENOP LDU,     2   8E GENOP LDX,
2 108E GENOP LDY,     0   4E GENOP JMP,     0   8D GENOP JSR,
0   48 GENOP LSL,     0   44 GENOP LSR,     0   40 GENOP NEG,
1   8A GENOP ORA,     1   CA GENOP ORB,     0   49 GENOP ROL,

\ ------------------------------------------------------------
\   6809 assembler: general addr instr  (c) 29 03 85 BJR
\
0   46 GENOP ROR,     1   82 GENOP SBCA,    1   C2 GENOP SBCB,
0   87 GENOP STA,     0   C7 GENOP STB,     0   CD GENOP STD,
0 10CF GENOP STS,     0   CF GENOP STU,     0   8F GENOP STX,
0 108F GENOP STY,     1   80 GENOP SUBA,    1   C0 GENOP SUBB,
2   83 GENOP SUBD,    0   4D GENOP TST,

32 INXOP LEAS,  33 INXOP LEAU,  30 INXOP LEAX,  31 INXOP LEAY,

\ ------------------------------------------------------------
\   6809 assembler: branches            (c) 03 06 85 BJR
\
: CONDBR   <BUILDS (,)       \ opcode - | Conditional Branch
   DOES>   @  SWAP HERE 2+ -      \ addr -
   DUP 8BIT? IF  SWAP C, C,                    \ 8 bit
   ELSE  10 C, SWAP C, 2- W,  THEN RESET ;     \ 16 bit

: UNCBR   <BUILDS (,)        \ short:long - | Uncondit'l Bran
   DOES>   @  SWAP HERE 2+ -      \ addr -
   DUP 8BIT? IF  SWAP >< C, C,         \ 8 bit: use short opcod
   ELSE  SWAP C, 1- W,  THEN RESET ;  \ 16 bit: use long opcod

\ ------------------------------------------------------------
\   6809 assembler: branch instructions (c) 29 03 85 BJR
\
24 CONDBR BCC,  25 CONDBR BCS,  27 CONDBR BEQ,  2C CONDBR BGE,
2E CONDBR BGT,  22 CONDBR BHI,  24 CONDBR BHS,  2F CONDBR BLE,
25 CONDBR BLO,  23 CONDBR BLS,  2D CONDBR BLT,  2B CONDBR BMI,
26 CONDBR BNE,  2A CONDBR BPL,  21 CONDBR BRN,  28 CONDBR BVC,
29 CONDBR BVS,  2016 UNCBR BRA,  8D17 UNCBR BSR,

\ ------------------------------------------------------------
\   6809 assembler: conditions          (c) 03 06 85 BJR
\
24 CONSTANT CS  25 CONSTANT CC  27 CONSTANT NE  2C CONSTANT LT
2E CONSTANT LE  22 CONSTANT LS  24 CONSTANT LO  2F CONSTANT GT
25 CONSTANT HS  23 CONSTANT HI  2D CONSTANT GE  2B CONSTANT PL
26 CONSTANT EQ  2A CONSTANT MI  21 CONSTANT ALW 28 CONSTANT VS
29 CONSTANT VC  20 CONSTANT NVR

\ ------------------------------------------------------------
\   6809 assembler: structured cond'ls  (c) 03 06 85 BJR
\
: IF,     \ br.opcode - adr.next.instr 2  | reserve space
   C, 0 C, HERE 2 ;
: ENDIF,  \ adr.instr.after.br 2 - | patch the forward ref.
   2 ?PAIRS   HERE OVER -  DUP 8BIT? 0= 3 ?ERROR  SWAP 1- C! ;
: ELSE,   \ adr.after.br 2 - adr.after.this.br 2
   2 ?PAIRS   NVR C, 0 C, HERE SWAP  2 ENDIF, 2 ;
: BEGIN,  \ - dest.adr 1
   HERE 1 ;
: UNTIL,  \ dest.adr 1 br.opcode -
   SWAP 1 ?PAIRS   C,  HERE 1+ -  DUP 8BIT? 0= 3 ?ERROR  C, ;
: WHILE,  \ dest.adr 1 br.opcod - adr.after.this 2 dest.adr 1
   IF, 2SWAP ;
: REPEAT, \ adr.after.while 2 dest.adr.of.begin 1 -
   NVR UNTIL, ENDIF, ;
: THEN,  ENDIF, ;
: END,  UNTIL, ;

\ ------------------------------------------------------------
\   6809 assembler: code, ;code, ;c     (c) 14 11 85 BJR
\
FORTH DEFINITIONS ASSEMBLER
: ENTERCODE [COMPILE] ASSEMBLER   ASSEMBLER ALIGN  !CSP ;
: CODE CREATE ENTERCODE ;
: ;CODE  ?CSP COMPILE (;CODE) [COMPILE] [ ENTERCODE ;
   IMMEDIATE
ASSEMBLER DEFINITIONS

: ;C   CURRENT @ CONTEXT !   ?CSP   SMUDGE ;
: NEXT,   Y ,++ LDX,  X 0, [] JMP, ;
: NEXT    NEXT, ;
FORTH DEFINITIONS DECIMAL
```

```
GT greater
LO    lower                 PL    plus
LS    lower or same         ALW   always
LT    less than             NVR   never
MI    minus                 VC    overflow clear
NE    not equal/not zero    VS    overflow set
```

Infinite loops should use the 'NVR' (never true) condition code:

```
BEGIN,   ...   NVR UNTIL,
```

Internal Glossary and Description

This assembler was written before I acquired the habit of "shadow screen" documentation. So, I'll document all of the unusual words and features here. (Please refer to the program listing.)

The word **WITHIN** is a common Forth extension. The words **5BIT?** and **8BIT?** decide if a given value will fit in a 5-bit or 8-bit signed integer, so we can choose the correct indexed addressing mode.

The synonym (,) is defined because later I redefine , as an addressing mode.

ALIGN, et cetera, deserve some comment. This assembler was originally written for a metacompiler running on a 68000 system, which insisted upon word-aligning the DP after each Forth word was interpreted. This meant that any 6809 instruction which assembled an odd number of bytes would have a filler byte added—with catastrophic results! I fixed this by causing all of the assembler words to do the aligning themselves. Thus, the 68000 Forth never inserted any filler, and I always knew when the DP had been adjusted. **HERE** and **C,** were redefined accordingly.

The word **W,** allows 6809 word operands to be compiled on either big-endian or little-endian host machines. **><** is a byte-swap operator provided by the Forth I used.

Some 6809 instructions have a one-byte opcode, and some have two bytes. Opcodes are stored as a 16-bit value. If the high 8 bits are nonzero, they are the first opcode byte. **OPCODE,** lays down one or two bytes, accordingly.

The **MODE** variable indicates whether the addressing mode is Immediate, Direct, Indexed, or Extended. **#** and **<>** set the first two modes; Extended is the default when no mode is set. (This is the first addressing-mode technique described in the previous article.)

The Indexed addressing modes in the 6809 add a "postbyte" to the opcode, which contains mode information and a register number (for X, Y, U, or S). **INDEXREG** puts the two-bit register number into the postbyte, and also sets **MODE**. Note that the postbyte is passed on the stack. (This is the second addressing-mode technique.)

XMODE defines the "simple" Indexed modes. These modes each have a fixed postbyte, modified only by the register number (as supplied by **INDEXREG**). For example, the word **,++** fetches the postbyte 81 hex and then invokes **INDEXREG** to insert the two register bits.

The word **,** rearranges the stack and builds the postbyte for the constant-offset Indexed mode.

The word **,PCR** provides the postbyte for the PC-relative Indexed mode. Since this has no register operand, **INDEXREG** isn't used. **MODE** must be explicitly set to 20 hex.

The word **[]** indicates indirection. For the Indexed addressing modes, this is done by setting the "indirect" bit in

Figure 1: **Forth and Motorola Assemblers Comparison Chart**

This chart shows the Forth assembler's equivalent for all of the Motorola assembler instructions and addressing modes. This is not an exhaustive permutation of addressing modes and instructions; it is merely intended to illustrate the syntax for all possible addressing modes in each instruction group.

Refer to a 6809 data sheet for descriptions of allowable operands, operand ranges, and addressing modes for each instruction.

Instructions which require two operands (TFR and EXG) have the operands specified in the order: source, destination. (This is only significant for the TFR instruction.)

INDEXED ADDRESSING MODES

	–NON-INDIRECT–		–INDIRECT–		
TYPE	MOTOROLA	FORTH	MOTOROLA	FORTH	POSTBYTE
no offset	,r	r 0,	[,r]	r 0, []	1rri0100
5 bit offset	n,r	r n ,	defaults to 8-bit		0rrnnnnn
8 bit offset	n,r	r n ,	[n,r]	r n , []	1rri1000
16 bit offset	n,r	r n ,	[n,r]	r n , []	1rri1001
A-reg offset	A,r	r A,	[A,r]	r A, []	1rri0110
B-reg offset	B,r	r B,	[B,r]	r B, []	1rri0101
D-reg offset	D,r	r D,	[D,r]	r D, []	1rri1011
incr. by 1	,r+	r ,+	not allowed		1rr00000
incr. by 2	,r++	r ,++	[,r++]	r ,++ []	1rri0001
decr. by 1	,r-	r ,-	not allowed		1rr00010
decr. by 2	,r--	r ,--	[,r--]	r ,-- []	1rri0011
PC ofs. 8 bit	n,PCR	n ,PCR	[n,PCR]	n ,PCR []	1xxi1100
PC ofs. 16 bit	n,PCR	n ,PCR	[n,PCR]	n ,PCR []	1xxi1101
16 bit address	not allowed		[n]	n []	10011111

where n = a signed integer value,
r = X (00), Y (01), U (10), or S (11)
x = don't care

——INSTRUCTION SET——

Inherent addressing group

MOTOROLA	FORTH	MOTOROLA	FORTH
ABX	ABX,	NEGA	NEGA,
ASLA	ASLA,	NEGB	NEGB,
ASLB	ASLB,	NOP	NOP,
ASRA	ASRA,	ORA	ORA,
ASRB	ASRB,	ORB	ORB,
CLRA	CLRA,	ROLA	ROLA,
CLRB	CLRB,	ROLB	ROLB,
COMA	COMA,	RORA	RORA,
COMB	COMB,	RORB	RORB,
DAA	DAA,	RTI	RTI,
DECA	DECA,	RTS	RTS,
DECB	DECB,	SEX	SEX,
INCA	INCA,	SWI	SWI,
INCB	INCB,	SWI2	SWI2,
LSLA	LSLA,	SWI3	SWI3,
LSLB	LSLB,	SYNC	SYNC,
LSRA	LSRA,	TSTA	TSTA,
LSRB	LSRB,	TSTB	TSTB,
MUL	MUL,		

Register-register group

MOTOROLA	FORTH	MOTOROLA	FORTH
EXG s,d	s d EXG	TFR s,d	s d TFR

Immediate-addressing-only group

MOTOROLA	FORTH	MOTOROLA	FORTH
ANDCC #n	n # ANDCC,	PSHS regs	n # PSHS,
CWAI #n	n # CWAI,	PSHU regs	n # PSHU,
ORCC #n	n # ORCC,	PULS regs	n # PULS,
PULU regs	n # PULU,		

Note: Motorola allows the PSH and PUL instructions to contain a register list. The Forth assembler requires the programmer to compute the bit mask for this list and supply it as an immediate argument.

Indexed-addressing-only group
(with example addressing modes)

MOTOROLA	FORTH	MOTOROLA	FORTH
LEAS D,U	U ,D LEAS,	LEAX [,S++]	S ,++ [] LEAX,
LEAU -5,Y	Y -5 , LEAU,	LEAY [1234]	1234 [] LEAY,

General-addressing group
(with example addressing modes)

MOTOROLA	FORTH	MOTOROLA	FORTH
ADCA #20	20 # ADCA,	LDB <30	30 <> LDB,
ADCB <30	30 <> ADCB,	LDD 2000	2000 LDD,
ADDA 2000	2000 ADDA,	LDS [1030]	1030 [] LDS,
ADDB [1030]	1030 [] ADDB,	LDU ,X	X 0, LDU,
ADDD ,S	S 0, ADDD,	LDX 23,Y	Y 23 , LDX,
ANDA 23,U	U 23 , ANDA,	LDY A,S	S A, LDY,
ANDB A,X	X A, ANDB,	LSL B,U	U B, LSL,
ASL B,Y	Y B, ASL,	LSR D,S	S D, LSR,
ASR D,X	X D, ASR,	NEG ,X+	X ,+ NEG,
BITA ,S+	S ,+ BITA,	ORA ,S++	S ,++ ORA,
BITB ,X++	X ,++ BITB,	ORB ,U-	U ,- ORB,
CLR ,Y-	Y ,- CLR,	ROL ,Y--	Y ,-- ROL,
CMPA ,U--	U ,-- CMPA,	ROR 12,PCR	12 ,PCR ROR,
CMPB -5,PCR	-5 ,PCR CMPB,	SBCA [,U]	U 0, [] SBCA,
CMPD [,Y]	Y 0, [] CMPD,	SBCB [7,U]	U 7 , [] SBCB,
CMPS [7,Y]	Y 7 , [] CMPS,	STA [A,X]	X A, [] STA,
CMPU [A,S]	S A, [] CMPU,	STB [B,Y]	Y B, [] STB,
CMPX [B,U]	U B, [] CMPX,	STD [D,S]	S D, [] STD,
CMPY [D,X]	X D, [] CMPY,	STS [,U+]	U ,+ [] STS,
EORA [,Y+]	Y ,+ [] EORA,	STU [,Y++]	Y ,++ [] STU,
EORB [,U++]	U ,++ [] EORB,	STX [,S-]	S ,- [] STX,
COM [,S-]	S ,- [] COM,	STY [,X--]	X ,-- [] STY,
DEC [,X--]	X ,-- [] DEC,	SUBA [3,PCR]	3 ,PCR [] SUBA,
INC [5,PCR]	5 ,PCR [] INC,	SUBB [300]	300 [] SUBB,
JMP [300]	300 [] JMP,	SUBD 1234	1234 SUBD,
JSR 1234	1234 JSR,	TST #2	2 # TST,
LDA #20	20 # LDA,		

Note that, in the Forth assembler,
signifies Immediate addressing, and
<> signifies Direct addressing.

Many instructions do not allow immediate addressing. Refer to the Motorola data sheet.

Branch instructions

MOTOROLA	FORTH	MOTOROLA	FORTH
BCC label	adrs BCC,	BLT label	adrs BLT,
BCS label	adrs BCS,	BMI label	adrs BMI,
BEQ label	adrs BEQ,	BNE label	adrs BNE,
BGE label	adrs BGE,	BPL label	adrs BPL,
BGT label	adrs BGT,	BRA label	adrs BRA,
BHI label	adrs BHI,	BRN label	adrs BRN,
BHS label	adrs BHS,	BSR label	adrs BSR,
BLE label	adrs BLE,	BVC label	adrs BVC,
BLO label	adrs BLO,	BVS label	adrs BVS,
BLS label	adrs BLS,		

The branch instructions in the Forth assembler expect an absolute address. The relative offset is computed, and the "long branch" form of the instruction is used if necessary.

"Did you know that if a beaver two feet long with a tail a foot and a half long can build a dam twelve feet high and six feet wide in two days, all you would need to build the Kariba Dam is a beaver sixty-eight feet long with a fifty-one foot tail?" —Norton Juster

the postbyte. (This is an example of the third addressing-mode technique.) For the Direct addressing mode, [] must change the mode to Indexed and supply the Extended Indirect postbyte.

Register definitions are simple CONSTANTs. Note that the synonyms W, IP, RP, and SP are defined for the registers X, Y, S, and U. I use these synonyms when writing Forth kernels.

INHOP defines the Inherent (no operands) instructions.

IMMOP defines the Immediate-only instructions. These all expect a byte operand, and they check to make sure that the # addressing mode was specified (MODE=0).

RROP defines the register-register instructions, TFR and EXG. Note that it doesn't check that both operands are the same size; the programmer is presumed to know better.

+MODE is a fudge. All of the general-addressing instructions form their opcodes by adding 0, 10, 20, or 30 hex to a base value, *except* those instructions whose Direct opcode takes the form 0x hex. These instructions use 6x for Indexed mode, and 7x for Extended, so the assembler assumes a "base opcode" of 4x, and adds 10, 20, or 30 hex. (There is no Immediate mode for these instructions.) Then, if the resulting opcode is 5x, we know that Direct mode was specified, and the opcode should really be 0x hex. +MODE applies the MODE value to the base opcode; it then checks for 5x opcodes and changes them to 0x.

PCREL lays the postbyte and operand for PC-relative addressing. If the signed offset fits in 8 bits, the postbyte is modified for the short form and a single-byte operand is used. Otherwise, the long form postbyte and two-byte operand is used.

NOTINDIR? checks the indirection bit in the postbyte.

COFSET lays the postbyte and operand for constant-offset Indexed addressing. If the offset fits in 5 bits, and indirection is not used, then the postbyte is modified for the 5-bit offset. Otherwise, if the offset fits in 8 bits, the postbyte is modified for the 8-bit form (single-byte operand); if the offset requires 16 bits, the long form is used (two-byte operand).

EXTIND lays the postbyte and operand for Extended Indirect addressing.

INDEXED lays the postbyte and (if required) operand for all of the Indexed modes. COFSET, PCREL, and EXTIND are the special cases which require operands; a CASE statement identifies these by the postbyte value. All other postbyte values are "simple" modes which have no operands; they are handled in the default clause of the CASE statement.

Some 6809 Immediate instructions require an 8-bit operand, and some a 16-bit operand. Others don't allow Immediate mode. IMMED handles this by testing the second parameter in an opcode word defined by GENOP, and laying one or two operand bytes, accordingly. If "zero" operand bytes are indicated, this means that this addressing mode is invalid with this instruction.

GENOP defines the instructions which can have any addressing mode. It selects one of four actions depending on MODE. As noted above, the parameter "immedsize" can be used to disallow Immediate mode for any instruction.

INXOP defines the instructions which can have Indexed addressing mode. It checks that the MODE value is correct (20 hex) before assembling the instruction.

CONDBR and UNCBR build the branch instructions. They are identical, except in how they modify the instruction when the long form is required. CONDBR makes the long form by prefixing a 10 hex byte; UNCBR makes the long form by substituting an alternate opcode (both opcodes are stored in a 16-bit value). Both of these words take an absolute address, and compute the relative offset from the branch instruction; the short or long form of the branch is then automatically chosen.

CS through NVR are condition code CONSTANTs for the structured conditionals. These are actually the opcodes which must be assembled by the conditionals such as IF, and UNTIL,. As noted in the previous article, IF, and UNTIL, must actually assemble the logical inverse of the stated condition; this is handled in the 6809 assembler by defining each of these constants to contain the "inverse" opcode. For example, the constant CS (carry set) actually contains the opcode for a BCC instruction. This is because the phrase CS IF, must assemble a BCC.

Structured conditionals were described in detail in the previous article. Since the condition codes are in fact opcodes, the requisite conditional jumps can be assembled directly with C,. Also, these conditionals use the fig-Forth "compiler security": each conditional leaves a constant value on the stack, which much be verified (by ?PAIRS) by its matching word. For example, IF, leaves 2 on the stack; ELSE, and ENDIF, (a.k.a. THEN,) do a 2 ?PAIRS to resolve this.

ENTERCODE CODE and ;CODE are, of necessity, "tuned" to a particular Forth implementation. For example, since this fig-Forth model's CREATE automatically sets up the code field properly, it's not necessary for CODE to patch the code field to point to the new machine code. Other Forth models have a different assumption, so some phrase such as

```
CREATE  HERE 2- !
```

will no doubt be needed to set the code field pointer.

ENTERCODE (and thus CODE and ;CODE) also use !CSP to check for stack imbalances. This means that each CODE definition must end with ;C (which uses ?CSP to resolve !CSP). ;C also unSMUDGEs the Forth word being defined. These are also fig-Forth usages, which may not apply to your Forth model.

Finally, NEXT, is an example of a simple assembler macro. It assembles the two-instruction sequence which is the inner interpreter (NEXT) of an indirect-threaded 6809 Forth. NEXT is a synonym.●

Local Area Networks

Bridges and Routers

By Wayne Sung

Ethernet has a specified physical distance limit due to the timing mechanisms in use, but even those types of networks which are less dependent on timing will have bounded sizes. This is mostly due to signal losses in the physical wiring.

To extend a network to a distant place, it will be necessary to have both a delivery system, such as the broadband system discussed last time, and a device which is built specifically for extending networks. Two types of devices often used to accomplish this are *bridges* and *routers*. Both operate on the assumption that only a small proportion of the traffic on a LAN is meant to be sent to a distant point.

In either case, the device can be viewed as having two or more "sides." One of these is an Ethernet and the others may be point-to-point lines or just as easily more Ethernets.

Bridges

Bridges are the simpler device. In most cases, a bridge tries to learn what devices are on each of its sides. It does this by examining every packet that passes by, and remembering the source addresses.

Having built such a table, it looks at the destination address of a packet. If this address exists in the table, there is no further need to process the packet because it is "local." If the destination address is not local, then the packet gets passed to another side. This action is called filtering.

During the packet transition it is possible to apply some administrative conditions to prevent the packet from going across. This is called custom filtering. For example, you could filter out all broadcasts.

Since bridges function only according to Ethernet addresses, there is no sensitivity to different types of software that might be in use. Thus bridges are very easy to set up, and generally work correctly.

However, the software must understand that it is possible to have networks larger than a single Ethernet. The packet delivery time within a single Ethernet is quite small. If two Ethernets are joined by bridges, even if operating at T1 rates (1.5 Mb/s), there will necessarily be about a 6 to 1 slowdown.

This extra delivery time often causes trouble, because the software was not expecting it and once again we have a retransmission problem—the software thinks packets are lost when actually they simply have not arrived yet.

Another drawback of bridges is that they don't normally do anything about broadcasts and multicasts, since by definition these are supposed to go everywhere. It is possible to use filters to cut out some of these, but this requires you know something about the packets.

A severe problem with bridges is the possibility of loops. If I took a bridge with one Ethernet port and one serial port, connected its serial output to its serial input, and then connected the Ethernet, what would happen?

The easiest case to see would be a broadcast. It goes in the Ethernet port, passes out the serial port, returns to the input side of the serial port, and ends up right back on the Ethernet.

Why would I do something like this? Well, I don't do it on purpose! However, sometimes transmission equipment fails and causes a loop condition. Otherwise someone might be testing an adjacent line and inadvertently plugs into mine.

Since the serial line is considerably slower than the Ethernet, a duplicate packet is quite possible. Many types of software will fail with duplicate packets. In another case, the address becomes learned on more than one side of a bridge, causing that address to lose communication with the distant side.

This last case can cause a lot of problems, simply because it's not what you would expect. Let's look at a two-port bridge. The two sides are called A and B, and host address 1 (which is on side A) has been learned on both sides because of a loop.

If host 1 sends to host 2 (which is on side B) the packet will go through fine. When host 2 replies, the B side of the bridge will match the destination address (host 1) in its table and not pass the packet through. Functionally host 1 cannot talk to anyone off side A.

This loop problem also prevents using multiple links for redundancy without some extra effort. Most solutions to this problem take the form of putting one in standby or splitting traffic by type across several links.

Two terms you will encounter in the bridge numbers war are the "filtering rate" and the "forwarding rate."

Since a bridge has to examine every packet to decide whether to take any action on it (as well as to learn the addresses) it pays to have a high filtering rate, which is the rate at which the bridge is able to watch packets pass by. Note that this does not include custom filtering.

Given that a packet should indeed be passed to another side, the rate at which packets can be continuously sent through is called the forwarding rate. Custom filtering often reduces forwarding rate, and can affect filtering

Wayne Sung has been working with microprocessor hardware and software for over ten years. His job involves pushing the limits of networking hardware in attempting to gain as much performance as possible. In the last three years he has developed the Gag-a-matic series of testers, which are meant to see if manufacturers meet their specs.

rate if address matching is done in software.

Even though you normally want a high filtering rate, forwarding rate is less important. If you need forwarding rates exceeding 20% of one Ethernet you should try to rearrange the layout so that more traffic can stay local. If a large amount of traffic is passing through a bridge then it's not doing you much good.

Also, in those cases where a bridge connects to a serial link to go to some distant location, there is no point having forwarding rates higher than the serial side can handle. A T1 line, for example, can only handle about 2000 minimum size packets per second.

Some vendors have taken the number of interfaces in their units and multiplied the total rates by the number of interfaces. This gives impressive but unrealizable numbers. Each Ethernet can have at most 14880 packets per second. If either rate is quoted as higher than that there is a problem in the way the number is generated.

The fallacy is that when a packet is forwarded it has to occupy more than one interface and so the numbers cannot simply be added together. There is also the problem of what mix of packets is being watched. If the same addresses are in all of them there won't be much work for the bridge. If a continuously changing set of addresses is present the bridge will spend a considerable amount of time building its address tables and will slow down.

Routers

A router looks and connects the same way as a bridge, but acts only on packets it recognizes. This is called "protocol specific" operation, and setting up a router definitely requires that you know something about the protocol you are using.

Earlier routers handled only one or two protocols, but this is no longer true. However, not all protocols are capable of being routed. Fundamental to router operation is the necessity of a protocol having an address structure that allows inter-networking (more specifics on this next time).

Since a router looks at some detail into a packet, it can avoid the bridge loop problem. Packet forwarding is now done based on protocol specific addresses rather than Ethernet addresses and so it can immediately be determined which addresses are local and which are not. A looped packet will have a local source address and can be discarded immediately.

A router also solves the broadcast problem to a large extent, because most broadcasts are of a local nature and are not passed through.

Finally, routers generally allow the use of not only multiple links, but essentially an arbitrary set of links across a very large number of routers. This would be much more difficult to get right with a system of bridges. This is also because packets do not get looped with multiple links.

Note, though, that having multiple links between two routers does not guarantee they can provide total throughput beyond what one link provides. This is very dependent on the exact protocol in use. Having multiple links does allow automatic reconfiguration when one link fails.

Routers have a different kind of looping problem, though not normally visible to users. This is known as a routing loop. When a router has knowledge of the existence of a network, it is said to have a route to that network. This knowledge is generally sent from one router to another, and each router will merge the knowledge of its own networks and those that can be reached remotely.

When a router has only one remote link, it is just as well to say all networks are through this one link. This is called using a default route.

One easy way to cause a loop, then, is to connect two units together where each defaults to the other.

In the case where you get a data packet for a destination not local to you, you promptly send it away via the default route. This causes the other unit to send it back again, because it thinks you're the proper way to handle it. This continues until some other mechanism decides this has gone on long enough and kills the packet.

Many protocols have a trip counter, which is to be decremented each time a packet passes a router. This is to prevent packets from looping forever. With the rather high speed equipment available today, this becomes very important because one looping packet could use up an entire link.

Another possible cause of a routing loop is a problem similar to an unstable feedback control mechanism. The routing knowledge that gets passed among units can suddenly change. For example, a line goes down causing a network to no longer be reachable. Although this state change will eventually be passed along, in general routers are set up not to respond to changes too quickly because many of these are transient.

For the amount of time it takes the state change to be fully propagated into the system of routers, there is a potential routing loop. Let's say I just lost a network. I stop telling my neighbor that I have that network. Some time later the neighbor will realize I have not been announcing that network and will then delete that knowledge.

Until then, however, it will continue to send packets destined for that network to me. However, since I have lost knowledge of that network I will send them back via my default, which in this case happens to be my upstream neighbor. It sends them back to me. And so it continues until the trip counter expires.

It has traditionally been easier to stop announcing a network than to explicitly announce it as unreachable, although there are certainly other ways to do it. There is much ongoing study concerning convergence in routing protocols.

Filtering rates do not apply to routers, because they do not read every packet. Only those packets specifically addressed to them are received and processed. You can also set custom filters in routers to further limit packet forwarding.

Forwarding rates do apply, and again it makes no sense to have forwarding rates that exceed any link's capability to handle that rate.

By the way, some readers will probably have caught the tradeoff between packets-per-second and bits-per-second. Since the maximum bit rate of an Ethernet is fixed, larger packets will result in fewer packets per second. This is usually advantageous, because this means lower interrupt rates in the packet handlers.

This is why larger packets are said to be more efficient (but only if they are actually full of data). There is one kind of traffic where you cannot store up enough data to make a large packet before sending it - terminal traffic. Here you want the quickest response possible, so even a minimum sized packet would have some unused space.

Software response rates often limit the number of packets

See LAN, page 18

Advanced CP/M

I/O Redirection in CP/M Plus

By Bridger Mitchell

Perhaps this column should be titled "The Return of the Oxymoron." After all, nearly all personal computer users outside the CP/M and Z community would suspect anyone who juxtaposes "advanced" and "CP/M" of brain damage! Nevertheless, I'm glad to be resuming the *Advanced CP/M* column after an absence of several issues.

I'll be focusing on CP/M Plus, also known as CP/M 3, the last general-purpose operating system for the 8080 processor from Digital Research. We'll cover several topics in this and the following columns to highlight features introduced in CP/M Plus and significant differences from the more familiar CP/M 2.2 version. This time we'll examine the redirection of input and output and aspects of banked memory. A later column will take up how the Z-System was ported to CP/M Plus—some of the key design decisions that are built into Z3PLUS. Since these topics are necessarily complex and interrelated, I'll also refer to other *TCJ* articles at several points.

I/O Redirection

The UNIX operating system is justly famous for its ability to effortlessly redirect both input and output from the console. By simply typing "prog < in_file" at the command prompt the user can cause what would normally be the keyboard input to come instead from a file; all that is required is the redirection symbol "<" followed by the name of a suitable file. Similarly, he can redirect the output of the program into a file with the command "prog > out_file", thus capturing everything that would otherwise be output to the terminal's screen.

At the heart of the UNIX design is the decision to treat character (unit-record) and mass-storage (block) devices identically when making operating system calls. Thus, you "read(...)" a byte from a file, and you also "read(...)" a byte from the console. The code is identical code—the actual source of the data is determined by what device (console or file) has been previously opened. So, when the unix shell (command processor) sees "<" on the command line it simply changes the open device and otherwise executes the same code for processing character input.

The CP/M 2.2 operating system has nothing like this flexibility. There are separate system calls for each type of device (console input, console output, file output, ...). And there is no provision for redirection of output. With some difficulty, some redirection of input is possible, using the SUBMIT and XSUB utilities or the more flexible ZEX in a CP/M 2.2 Z-System. Also, under the Z-System with the more advanced BDOS provided by ZSDOS or ZRDOS, plus a suitable Z-System I/O Package, console output can be redirected to a file. And somewhat more general output redirection is available under BackGrounder ii with standard CP/M 2.2.

CP/M Plus made some significant improvements on redirection. Like CP/M 2.2 it also maintains separate system calls for individual types of devices, but it builds hooks into the operating system to make redirection possible. Submit files can be automatically executed by typing at the command prompt the name of a script file with type "SUB"; they can also be invoked in CP/M-2 fashion by typing the command "SUBMIT" followed by the name of the script file.

CP/M Plus provides two general-purpose redirection utilities—GET.COM and PUT.COM. Executing GET with an appropriate command line causes the command processor and programs to get their console input from a specified script file. Executing PUT shunts the console output into a file. (GET.COM should not be confused with the Z System built-in GET command, which loads a file to a specified address in memory.)

We'll examine how CP/M Plus implements this redirection a little later. First, however, let's look at how banked memory affects the operating system.

Banked Memory

CP/M Plus gives the user a larger program area (TPA) than is available in CP/M 2.2. It does this by moving much of the operating system and its data structures into alternate memory. *Figure 1* sketches the memory map. The always-resident portion of the operating system is at the very top of memory. Below it somewhere, perhaps at 0F000h or 0F800h, alternate banks of memory can be active. Conventionally, "bank 0" contains the rest of the operating system, and "bank 1" holds the user program and data. In addition, other banks may be used for disk buffers, a copy of the CCP, et cetera. In many implementations the code portions of the

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

*Bridger can be reached at Plu*Perfect Systems, 410 23rd Street, Santa Monica CA 90402 or at (213) 393-6105 (evenings).*

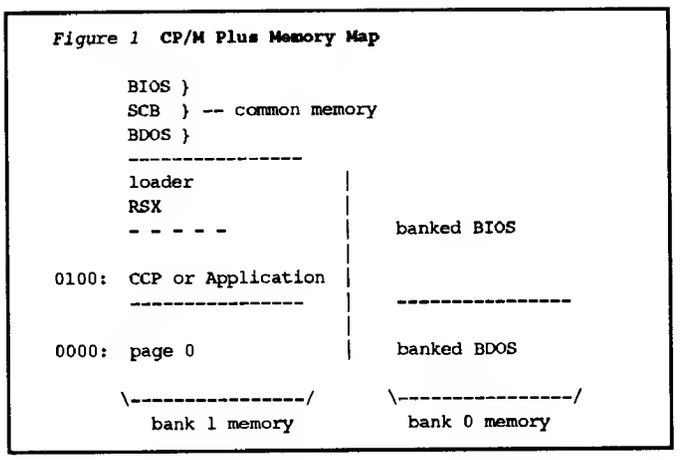
Table 1 **CP/M 3 System Control Block (SCB)**
(** = CCP105 additions, zz = Z3PLUS additions)

hex offset	hex off-80h	var. name, bits	description
These vectors called by the banked BDOS. GET and PUT RSXs redirect IO through them by setting "JP addr" to "LD HL,rsx_addr", ensuring bank 1 (TPA) access.			
68			jp ?wboot
6b			jp execute: set bank 1, .. jp (hl), set bank 0
6e			jp ?const
71			jp execute
74			jp ?conin
77			jp execute
7a			jp ?cono
7d			jp execute
80			jp ?list
83			jp execute
86-8f			spares? The extended status .. vectors were never .. implemented here.
90-91	10-11	@STAMP	stamped drive vector
92-92	12-13	@RELOG	relog drive vector
93-97			spare?
98-99	18-19		resident BDOS addr
98-9b			spare?

Start of 100 byte (decimal) SCB; address of structure at this point is returned by BDOS function 49.			

9c	1c	@HSHCK	Hash check byte.
9d	1d	@HSHDRV	Drive compare byte.
9e-9f	1e-1f	@HSHNAME	Filename code
a0	20	@HSHEXT	Extent code byte.
a1	21	@VERSION	CP/M version number
a2			spare?
a3			spare?
a4	24	**	
		b0-b4	submit file user number (+1; .. 0=use default)
		b5	reset-disk system flag
		b6	erase submit file flag
		b7	display named dir. flag
a5	25	**	
		b0-b4	user number (+1) for loader .. (0=use default)
		b5	unused?
		b6	PRL file flag, for loader
		b7 **, not zz	LBR file flag, for loader
ab	2b	--	submit drive
ac-ad	2c-2d	@RTNCODE	BDOS return code word
ae	2e	--	next command page (multiple .. command line RSX page)
af	2f	--	CCP drive (0=A)
b0	30	--	CCP user number (0=zero)
b1-b2	31-32	--	ptr to next cmd if multiple .. command
b3	33	@CHAIN	program chain flags
		b0	submit flag (file search .. found '$')
		b1	RSX flag - delete inactive .. RSX's
		b6	change default U/D to last .. program
		b7	chain flag
b4	34	@SVBUF	Flag saying to use previous .. line buffer.
		b0-b1	display DU: when loading .. file
		b3-b4 zz	Z3PLUS search order: 00: COM .. only, 01: COM, SUB, .. 10: SUB, COM, 11: PRL, .. COM ** .. (DRI uses: 00: COM, 01: .. SUB, 10: PRL)
		b5 ??	reset disks
		b6	1=use default page mode
		b7	CCP is executing flag (used .. by ^W edit fn.)
b5	35	--	
		b1	auto-start command done flag
b6	36	@WIDTH	Console width byte
b7	37	@COLUMN	Console cursor column number
b8	38		console page length
b9			spare?
ba-bb	3a-3b	@BUFPTR	if NZ, ptr to redirected .. console input
bc-bd	3c-3d	@POINTR	if NZ, ptr to next line of .. redirected input
be-bf	3e-3f	@CIVEC	bit mapped console input .. phys. devices
c0-c1	40-41	@COVEC	bit mapped console output .. phys. devices
c2-c3		@AIVEC	bit mapped console aux. .. input phys. devices
c4-c5		@AOVEC	bit mapped console aux. .. output phys. devices
c6-c7		@LOVEC	bit mapped list output phys. .. devices
c8	48	**	current page mode (0=page .. pause)
c9	49	**	default page mode
ca		@CTRLH	Control-H active flag.
cb		@RUBOUT	Rubout active flag.
cc	4c	@KEYST	submit mode keyboard status .. byte
cd	4d		spare?
de	4e		spare?
cf-d0	4f-50	@MODE	Console mode bytes.
		b0	fn 11 true only on ^C
		b1	no ^S-^Q, TAB, ^P
		b2	no TAB, ^P
		b3	no ^C action
		b8-b9	redirection status: 00: .. conditional, 01: false, .. 10: true, 11: no .. redirection
d1-d2	51-52	@BNKBF	
d3	53	@DELIM	Print string delimiter.
d4	54	@OUTFLG	List output flag.
d5	55	@KEYLK	Keyboard lock byte.
d6-d7	56-57	scb	addr of this (100-byte only) .. SCB structure (=xx9c)
d8-d9	58-59	@CRDMA	Current DMA address
da	5a	@CRDSK	BDOS current disk
db-dc	5b-5c	@VINFO	(de) registers on entry to .. this BDOS
dd	5d	@RESEL	Flag byte indicating a file .. i/o function
de	5e	@MEDCHG	Media change flag byte
df	5f	@FX	BDOS function number
e0	60	@USRCD	BDOS current user number
e1-e2	61-62	@ENTRY	Main file position entry .. storage
e3-e4	63-64	@HOLDFCBH	
e5		@MATCH	char. count for matching .. filenames
e6	66	@MLTIO	Multi-sector count byte
e7	67	@ERMDE	Error mode flag byte
e8-eb	68-6b		drive search chain (0=def, .. ff = end of chain)
ec	6c	--	drive for temp. file
ed	6d	@ERDSK	disk with error

Figure 1 **CP/M Plus Memory Map**



banked operating system are in read only memory. In what follows I will assume for this discussion that bank 0 extends from 0000h to EFFFh and that the address range F000h to FFFFh is common to both banks.

The CP/M Plus resident BDOS manages the activation of the appropriate memory, switching in the correct bank before accessing it. It does so by calling one of the extended BIOS calls for memory bank switching, and this process is entirely transparent to a user application. In fact, as we are about to see, a normal application program should never attempt to manipulate banked memory!

The flow of an ordinary BDOS call to get a character from the console looks something like this, where the level of indentation indicates the level of subroutine nesting:

```
0100:           ld   c,1
0102:           call BDOS (0005)
  0005:           jp resident-BDOS (at F600)
  F600:           call bank-switch and return
  F603:           call console input (at 0200)
    0200':          call BIOS console input
      F800:           BIOS console input driver ..
                      ret
    0203':          etc.
                    ret
  F606:           call bank-switch and return
  F609:           ret
0105:           process returned char, etc.
```

I've marked addresses in bank 0 with a prime symbol. Between the two calls to the bank-switching function the user's program will be "invisible," and during that time the banked BDOS, with its console input function at 0200', will be available to be called.

hex offset	hex off-80h	var. name, bits	description
ee	6e		spare?
f0		@MEDIA	possible media change flag .. (door open)
f1			spare?
f2			spare?
f3	73	@BFLGS	BDOS flags.
		b6	double alv's
		b7	expanded error messages
f4-f5		@DATE	julian date.
f6		@HOUR	hour, bcd
f7		@MIN	minute, bcd
f8		@SEC	second, bcd
f9-fa	79-7a	@COMMON	addr of common memory (0000 .. if not banked sys.)
fb-fd		@ERJMP	jmp to error instr. in .. banked BDOS
fe-ff	7e-7f	@MXTPA	top of tpa+1/lowest RSX .. (protected memory addr)

Actually, this pseudocode fragment contains a disastrous "gotcha." If the user program has set a stack located in the user program area of memory, the return address pushed onto the stack by the call to the bank switching routine will be invisible. On return from that routine the processor will pop a garbage value off whatever address the stack pointer is now pointing to in bank 0 and "return" to continue execution at that address. So, the resident BDOS must save the stack pointer and set a safe stack in permanently-resident memory before calling the bank-switch routine. Before returning to the user program it must restore the original stack pointer.

The memory map in *Figure 1* shows most of the significant components of a CP/M Plus system. The System Control Block (SCB) is a 152-byte data structure that is used for intimate communication between components of the operating system. We will take up a few of the SCB items in this column and examine several others in the future. Refer to *Table 1*, which combines the information in Digital Research's CP/M Plus manuals, Jim Lopushinsky's additional documentation developed when he created CCP105 (an improved version of the DRI command processor), and my own discoveries made while designing Z3PLUS.

Resident System eXtensions (RSXs) are located in high memory, below the resident portions of the operating system. SUBMIT and GET commands both cause the GET.RSX to be loaded, and similarly the PUT command installs PUT.RSX.

Implementing Redirection by Indirection

When console input is redirected from a file, the characters will be read from the file, placed in a buffer within the GET RSX portion of memory, and then supplied one by one each time the console input system call is executed. CP/M Plus implements the following strategy. It replaces the BIOS console input vector, which normally points to the console input driver in the resident BIOS, with a vector pointing to a substitute input routine in the GET RSX. That routine will hand out a character from its buffer and do whatever housekeeping is required to refill its buffer periodically. When the end of the file is reached, the RSX will restore the original BIOS console vector, do some final housekeeping, and eventually vanish when overwritten by a new application.

Now, look back at the first code fragment. At 0200' the banked BDOS calls the resident BIOS for a character. But now, with the GET RSX installed, instead of vectoring to the console driver, control jumps to an address in the RSX. But hold on! The RSX may be in invisible memory, covered up by bank 0!

CP/M Plus's designers anticipated this potential catastrophe, and they avoid it by indirection. Instead of directly jumping to the BIOS console vector, they jump to a second vector of BIOS functions located at the start of the SCB. So, for console input the BDOS calls:

```
xx74  jp BIOS_conin
xx77  jp execute
```

But when the RSX is loaded this code is changed to:

```
xx74  ld hl,rsx_conin
xx77  jp execute
```

so that control first passes to this resident routine:

```
execute:    set bank 1
            push exec_ret address
            jp (hl)
exec_ret:   set bank 0
            ret
```

The result? Calls from the banked BDOS to console input switch in the user's bank, transfer control to the substitute conin routine in the rsx, then switch the operating system bank in again and return to the banked BDOS. (Again, of course, other code at the entry to the resident BDOS has ensured use of a stack that is always visible to all banks.)

If you look at the full SCB table you will see that there are similar indirection jumps for the warmboot, console status, console output, and list output functions.

File Access during Redirection

So far, CP/M Plus's designers have kept from crashing the system. But we're not done. Unlike CP/M 2's SUBMIT, which reads one file record at a time in the command processor, and XSUB and ZEX, which read the entire script file in advance and store it entirely in memory, CP/M Plus reads the script file as it needs it. That's nice, because only one record needs to be stored in memory, leaving a larger user program area.

But consider what would happen if your program were running on "automatic"—writing its output to a database file, say, while getting commands from a script instead of the keyboard? When 128 keystrokes have been supplied to the program, the SUBMIT RSX will have to read the next record from the script file. The file could quite easily be on a different drive and user number, and the RSX will necessarily have the BDOS read the data to a different DMA address. How can CP/M Plus read the script file and avoid damaging the user's file the next time his application writes to the database?

The solution was to have the RSX save the state of the file system before reading the script file and then restore that state before returning. The essential state data are kept in the SCB, beginning at xxD8 (see *Table 1*). The GET and PUT RSXs incorporate the necessary code to switch states between the primary process (the user's application program) and the subsidiary process (fetching console input from a file or writing screen output to a file).

When I designed ZEX 5.0 for use under Z3PLUS, it was an easy choice to make the redirection code into a CP/M Plus type of RSX. I also considered mimicking the approach of reading the script file as a subsidiary process. However, with version 5, ZEX provides for a rich range of control features specified by a syntax of full-word directives (see *TCJ* 38 and 40). In the CP/M 2.2 version, the script is compiled into a more compact, intermediate code that is stored in its entirity in the RSX's memory, and I used the identical method for the Z3PLUS version. Clearly, an alternative approach (perhaps for ZEX 6, anyone?) would be to write the intermediate code script to a file and have the RSX mimick the GET RSX by reading the file one record at a time.

New Applications of Redirection

Subsidiary-process access to the file system is a powerful capability that has not often been exploited. In order to create *BackGrounder ii* I had to develop a complete second- task capability that would leave any type of disk access that an application might make unaffected by background file activity. This, as it turns out, is a very stringent requirement — one that, for example, neither CP/M Plus nor Joe Wright's BPRINTER redirection utilities can fully satisfy. (They fail, for example, on programs such as DU that perform disk access by direct BIOS functions.) With this functionality in *BGii* it was then possible to write a spooler command that redirects console or printer output to a file and can print files asynchronously in the background while regular user programs are running in normal fashion.

An excellent use of CP/M Plus's secondary file access capability would be to write a background print spooler that runs as an RSX. The spooler would fetch data from a file, one record at a time, and spool it to the printer (list) device. The RSX itself can be quite small, leaving it to a separate application program to manage the details of the user interface, file opening, and the print queue.

A second interesting application would be a file transfer utility that would send or receive files through a modem port. This would allow the user to initiate a file transfer and let it proceed in the background while he proceeded to use other programs. Again, good design can keep the size of the resident code small.

The CP/M Plus operating system provides the advanced programmer with support for I/O redirection, secondary file access, and resident system extensions. Using these built- in tools, perhaps some of *TCJ*'s readers will take up the challenge of developing new applications. As always, I look forward to getting your feedback and ideas in the mail.●

LAN, from page 14

per second that can be handled, thus for smaller packets the number of bits per second that can be handled goes down.

> Spondulice is the Radix
> of Everything Pernicious
> —Michael Kaczmarek

Next Time

The largest internetwork in the world (outside of the telephone network) is the Internet, consisting of thousands of individual networks joined together. Typically a campus would have a LAN system of some sort and at one point there will be an internetwork router with one or more links to other routers. Wait! Isn't that the tandem switching arrangement mentioned earlier? Certainly.

Next time we will look more carefully at IP (the Internet Protocol) which is what runs across this world-wide Internet.●

ZCPR On a 16-Bit Intel Platform

By Brian Moore

A few issues back, Jay Sage brought up the idea of running ZCPR3 under a CP/M emulator. Since I've been using exactly such a system for some time, I'll pass on just what bringing it up has entailed.

The system I use for most of my work is a Vector Graphic model 4/40. This is a dual-processor system using an 8088 and a Z80 (similar to CompuPro's and Macrotech's S100 Dual Processor boards). In native mode, it runs CP/M-86 and to run Z80 programs, it uses a program called RUN8.CMD that sets up an artificial CP/M environment.

Having used CompuPro systems for many years, RUN8 was a familiar concept: much like CompuPro's SW! program. In fact, the same method described here should work on CompuPro's systems, as well as any other Z-80 CP/M emulator, even under other operating systems. I have also done a similar version for Vector Graphic's multi-user CP/M system. The concept should also work on Turbo-DOS systems, though that has yet to be tried. (I'm trying to do it under MS-DOS, too, but finding a good Z80 emulator has proven difficult.)

What does it take to bring up such a system? Surprisingly, not much! The only real requirement is an emulator that supports all the standard BIOS calls. Actually, even that isn't really necessary, if you want to do some extra work, but more on that later.

I'll take you through the evolutionary steps this system has gone through to get where it is now (full ZSDOS/ZCPR3 system) and this should answer your questions.

The Initial Problem

Under the original emulator provided by Vector Graphic, there was no command processor. If you gave an illegal command to the CP/M-86 CCP, it would look for a .COM and, if found, invoke RUN8 with the original command as the tail.

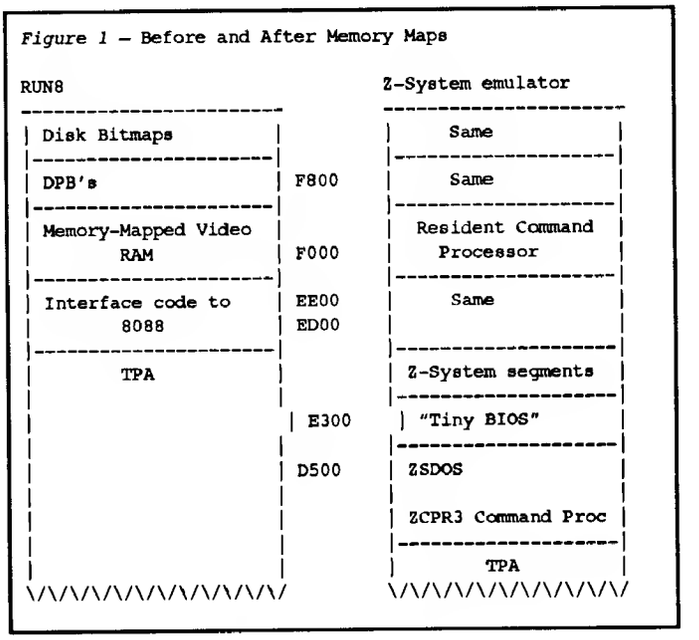
At this point it would set up an environment that looks somewhat like CP/M 2.2, load the requested program into that area and execute it. All the program's BIOS and BDOS calls would run through the emulator and be mapped into equivalent CP/M-86 calls.

The first version of the emulator had a serious bug: the BIOS jump table was not aligned on a page boundary. I checked the CP/M manual, and it never does say that the table must be page aligned, but many programs expect it to be.

I couldn't run many of my CP/M programs like this, including WS and anything written with SYSLIB (probably 99% of my software). I set about to write a little program that would move the jump table to a page boundary and then run the command specified as a parameter.

This worked as a partial solution, but this environment would disappear after each 8-bit program. I then set out to work around this.

Figure 1 – Before and After Memory Maps



Phase I : The Command Processor

On a regular CP/M-80 system the environment is set up at cold boot time, and certain maintenance of that environment is done with each warm boot.

I realized I needed to provide my own cold boot function, and to intercept the warm boot calls where I would do my own work, instead of returning to CP/M-86. This was done with a simple program I called (for brevity) X. X would first move the BDOS and BIOS entry points down in memory and space them 3.5k apart from each other. I found through bitter experience that some programs really don't like seeing the BDOS and BIOS jammed up next to each other as is common in emulated environments.

The BIOS was moved with a simple LDIR instruction and a patch to the

Brian Moore has been playing with Z80 software since 1978, when the computer addiction first bit. Since then, he has done hand-installations of ZCPR3.0 and 3.3 on several systems, and can't fathom why anyone would use Vanilla CP/M. Brian is the author of ARK11, the CP/M tool for creating ARC compatible archives. He can be reached on GEnie as BRIAN-CPM, The Machine BBS (503-747-8758) or at home (503-687-8531). His mailing address is 1048 Lincoln St, #1, Eugene, OR 97401.

Listing 1 — 86Z.COM

```
;
; 86Z—A Loader for ZCPR3 for the Vector 4. The ZCPR3 system
: image is loaded at 0D500h, and various parameters are
; intialized to appease just about any program that will run
; under ZCPR3.
;
; modified 9/19/89 for RUN8 v1.2
; modified 6/14/91 for Z80DOS24, which will let RUN8 act as
;   the BIOS ONLY. This should fix some of the quirks in
;   RUN8, and speed things up a bit.
; modified 7/25/91 for ZSDOS for better date handling.
;   Clock driver is loaded with pseudo-BIOS.
;
; Quirks in Vector's environment
RealBIOS    equ  0ee00h         ;Where the simulator's BIOS
NewBIOS     equ  0e300h         ;is a better place for it
TinyBDOS    equ  0d500h         ;low enough to save Z3
SimBDOS     equ  0ed06h         ;segments simulator's BDOS
Wboot       equ  NewBIOS + 36h  ;entry where we will put
                                ;the loader

            maclib  Z3BASE.LIB  ;get some constants

            .z80
            aseg

            org     100h

            ld      de,signon
            ld      c,9
            call    5

            ld      hl,CCP          ;what the heck,
            ld      de,CCP + 1      ; zero out the memory
            ld      bc,0ED00h - CCP
            ld      (hl),0
            ldir

            ld      hl,initcommand  ;copy the initial STARTUP
            ld      de,z3cl         ; command into the Z3
            ld      bc,initclen     ; command buffer
            ldir

            ld      hl,initpath     ;initialize the path
            ld      de,expath       ; (just need A0: for now)
            ld      bc,3            ; (someone else can set
            ldir                    ;  the rest)

            ld      a,0ffh          ;set wheel byte non-zero
            ld      (z3whl),a

            ld      hl,RealBIOS     ;set up our bogus BIOS
            ld      de,NewBIOS
            ld      bc,33h
            ldir

            ld      a,0c3h          ;Set up a jump to ZSDOS
            ld      (NewBIOS + 33h),a ;time routine
            ld      hl,TIME         ;
            ld      (NewBIOS + 33h + 1),hl

            ld      hl,mywboot      ;Copy in our artificial
            ld      de,wboot        ; warm boot code.
            ld      bc,chsize       ;
            ldir                    ;

            ld      c,0fh           ;open file
            ld      de,Z80DOSFCB    ;the Z80DOS binary image
            call    SimBDOS         ;(we assume it was found)

            xor     a               ;clear the record count
            ld      (Z80DOSFCB + 32),a

            ld      c,44
            ld      e,16            ;set multi-sector count
            call    SimBDOS

            ld      c,26            ;set the dma address
            ld      de,TinyBDOS     ; where to load Z80DOS
            call    SimBDOS

            ld      c,20            ;read sequential
            ld      de,Z80DOSFCB    ; (recs  0 - 15)
            call    SimBDOS         ;

            ld      c,44            ;Set multisector count
            ld      e,12            ; for next chunk
            call    SimBDOS         ;
            ld      c,26            ;
            ld      de,TinyBDOS + (128 * 16)
            call    SimBDOS         ;
            ld      c,20            ; read sequential
            ld      de,Z80DOSFCB    ;  (recs 16 - 27)
            call    SimBDOS         ;
            ld      de,Z80DOSFCB    ;
            ld      c,16            ; close our file
            call    SimBDOS         ;

            ld      hl,0EF7Fh       ; Set up scratchpad area
            ld      (0ef17h),hl     ;  (The simulator doesn't
            ld      (0ef27h),hl     ; do this for us so the
            ld      (0ef37h),hl     ; simple solution is to
            ld      (0ef47h),hl     ; fudge the entries so
                                    ; that a Z80 OS will be
                                    ; happy.)

; The following patch is to overcome a special quirk in the
; CP/M-80 simulator.  Instead of directly calling the
; simulator BIOS on this, we do it ourselves. We just patch
; our routine into the NewBIOS jump table.
            ld      hl,WRITE        ; Our WRITE replacement
            ld      (NewBIOS + 02aH + 1),hl

            ld      hl,wboot        ; Can't have a real WBOOT
            ld      (NewBIOS + 3 + 1),hl ; either

            xor     a               ; Force a log to A0: for
            ld      (4),a           ; no good reason

            jp      (hl)            ; load Z3 and execute the
                                    ; startup command...
                                    ; (force a warmboot)

signon:     db  0dh,0ah,'ZCPR 3.3 Loader for the Vector 4'
            db  0dh,0ah,'Version 1.00 by BEMoore',0dh,0ah
            db  0dh,0ah,'Please Stand By: Loading System.'
            db  '$'

initcommand:
            dw      z3cl + 4        ; pointer to first char
            db      z3cls           ; length of buffer
            db      0               ; offset into buffer
            db      'STARTUP;START'
            db      0               ; end of command

initclen    equ     $ - initcommand

initpath:   db      'A'-'@',0       ; force A0:
            db      0

Z80DOSFCB:  db      1,'Z80DOS  IMG'
            ds      25,0
;
; This is our own Mini-BIOS.  It serves mainly as a
; replacement for the WBOOT entry point to the BIOS, but it
; is also our opportunity to add some features and a fix to
; the BIOS.
mywboot:
            .phase  Wboot
            ld      sp,00feh        ;Get the stack out
                                    ;of harm's way

            ld      c,0fh           ;open file
            ld      de,z3binary     ;the binary image of Z3
            push    de              ;save this
            call    SimBDOS         ;(assume it was found)

            ld      c,44            ;set the multisector count
            ld      e,16            ;to 2k (the exact file
            call    SimBDOS         ;length)
```

new "Tiny BIOS" was made at location 0001. The BDOS, however, was only slightly more complex. X itself contained a copy of a "Tiny BDOS", and this was assembled with M80's PHASE directive. X would copy this to the proper location and patch the address at 0006 to point to this "Tiny BDOS".

The "Tiny BDOS" would pass all BDOS calls to the CP/M emulator except for function 0 (warm boot). This one function was trapped and would make a series of calls to the emulator to open and read a file containing the CCP, and then jump to it. The "Tiny BIOS" WBOOT jump went to this little routine, so a JP to location 0 would perform a warmboot.

The reason the above works, I realized, is because the CCP is just a special program, unique only that it's loaded by the BDOS and by its address.

Note that even at this stage I had, in effect, two BDOS's. One acts just like a CP/M 2.2 BDOS, reloading the CCP on function 0, and the other one drops back to CP/M-86 on function 0. But the "TinyBDOS" has a unique capability: it can do its reading by making BDOS calls (to the emulator) instead of the much lower-level BIOS calls.

This made it quite simple to read in the CCP at warm boot: I had all the standard CP/M BDOS calls for file I/O and even one for doing multi-sector reads. In effect, the BDOS could act like a transient program and call the "real" BDOS.

Phase II: A Real BDOS.

The above method had limitations: like trusting the emulator to behave properly (it has numerous bugs). For example, the BDOS input line function would echo an extra CR/LF, and would drop back to CP/M-86 on ^C. I also wanted to use programs like ZEX and BYE, which make direct patches to the BIOS. This would have odd effects, as the Tiny BDOS never called my Z80 BIOS, though some programs make direct BIOS calls.

I realized that the BDOS is much like the CCP: a special sort of program file loaded at a given point in memory. Actually, it's easier than the CCP in that the file need not be reloaded at warmboot time—just a few modifications to my 'X' program would give me a BDOS loader.

I used a copy of Z80DOS that I happened to have here—I assembled it for the proper address and saved it as a file called 'Z80DOS.BIN'. The CBOOT routine in the loader merely loads the BDOS in, then jumps to the WBOOT routine in the Tiny BIOS.

Of course, life is never that simple. Now that I was using a 'True' BDOS, I found that the BIOS emulation routines in RUN8 had some quirks, the worst of which was that calls to the WRITE routine would not properly flush a dirty buffer. Alternating drives, or just re-specifying the same drive, between calls to WRITE would introduce garbage into the buffer.

Without source code, I figured I had little chance of fixing this in the emulator. So, I wrote a replacement for the emulator's WRITE, or more precisely, an intercept for it. This relies on the information passed to the WRITE routine as to whether or not to do a pre-read, and whether an immediate flush is necessary. I force every call to WRITE to do a pre-read and immediate write.

There is a definite loss of speed with this method (and I'm still trying to figure out a better one), but the HD on this system has 256-byte physical sectors, so the overhead isn't that large. And actually, the Z80 is so much faster than the 8088 that by using a Z80 BDOS, I had a noticeable improvement in speed over the emulated version.

I also tried adding a SmartWatch to the system at this time, and found that the hardware wouldn't allow it. The Chip Select line to the ROM is *always* on, and the address lines always active, so it was impossible to pass the proper information to the SmartWatch. But, I realized that CP/M-86 has an interrupt driven clock built into it. My problem was finding it.

After some hacking at various system programs, I discovered how to manipulate the memory-manager (an address line latch, actually) to let the Z80 peek into the 8088's address space and read the clock.

This brings up an important point (and something I will be using when I get this ported to MS-DOS): the BIOS can deal with the system in any way it wants. For example, my WBOOT routine loads a disk file using the BDOS calls supported by the Emulator. That's much easier than having position-dependent files. The Clock routine works by peeking and poking into the other processor's memory.

While running under the emulator, then, the BIOS is also a special sort of program—capable of making BDOS calls! Just be careful to call the *emulator* BDOS, not Z80DOS or ZSDOS.

Phase III: Upgrading to ZSDOS

It didn't take too long using Z80DOS to realize I wanted the features of ZSDOS, but when I received it I realized it would involve another trick to install it.

ZSDOS is very easy to install on more normal systems. If you have MOVCPM or SYSGEN, it will figure out how to install itself into either of those. But I have no MOVCPM, since this system has never run real CP/M 2.2, and no SYSGEN since the system has no reserved tracks. (Actually, they are mapped out at a very low level of the OS.)

The trick was actually simple: I pretended I had SYSGEN and constructed in memory an image of what the boot tracks would look like if I had them. Z8E is real useful for this, since you can load the separate files at different addresses, and then move the BIOS jump table down just above them.

I saved this file out as ZSYS.IMG and pretended this was my SYSGEN image file. When I finished configuring ZSDOS, I returned to Z8E and stripped out the BDOS and saved it as its own file. It was so quick I saved it as Z80DOS.BIN, so I wouldn't have to change the filename in the loader—it does contain ZSDOS.

Additional Thoughts:

The trick to the whole operating system I'm using is that it is *all* considered by CP/M-86 to be one very strange transient program. As far as the CP/M-86 side goes, I make some BIOS and BDOS calls through the RUN8 program, and even-

---

"What's new?" is an interesting and broadening eternal question, but one which, if pursued exclusively, results only in an endless parade of trivia and fashion, the silt of tomorrow. I would like, instead, to be concerned with the question "What is best?" a question which cuts deeply rather than broadly, a question whose answers tend to move the silt downstream.

—Robert M. Pirsig

```
          xor     a               ;clear the record count
          ld      (z3binary + 32),a

          ld      c,26            ;set the dma address
          ld      de,ccp          ; where to load z3
          call    SimBDOS

          ld      c,20            ;read sequential
          pop     de
          push    de
          call    SimBDOS         ;file will now be loaded.

          ld      c,44
          ld      e,1             ;back to 1 record...
          call    SimBDOS

          di
          ld      a,0c3h
          ld      (0),a
          ld      (5),a

          ld      hl,TinyBDOS + 6 ;make the BDOS vector
          ld      (6),hl          ;point to the the
                                  ;"Tiny BDOS"

          ld      hl,NewBIOS + 3  ;set up the initial
          ld      (1),hl          ;bios area

          ld      a,(4)
          ld      c,a

          ei
          jp      ccp

; Replacement for RUN8's WRITE routine.
WRITE:    ld      c,1             ;force it to be
          jp      RealBIOS + 2Ah  ;"directory" write (ie,
                                  ;force buffer flush each
time)

z3binary: db      1,'Z33CPR  BIN'
          ds      24,0            ;rest of the fcb

; As long as we're in the BIOS,
; throw in a ZSDOS clock driver routine.
;
TIME:     ld      a,c             ;is command GET_TIME?
          or      a               ;
          jr      nz,noset        ;no -- go set the time

          call    clockon

          ld      hl,0f8c0h       ;ASCII clock data address
          ld      de,buffer       ;mm/dd/yy hh:mm:ss
          ld      b,6             ;Six fields to convert
getloop:
          ld      a,(hl)          ;
          and     0fh             ;Strip off ASCII bias
          add     a,a             ;Shift over to hi nybble
          add     a,a             ;
          add     a,a             ;
          add     a,a             ;
          ld      c,a             ;
          inc     hl              ;
          ld      a,(hl)          ;Get next digit
          and     0fh             ;Strip off ASCII Bias
          or      c               ;mask into hi-order
          ld      (de),a          ;and save it
          inc     de              ;
          inc     hl              ;skip to next
          inc     hl              ;Skip delimiter
          djnz    getloop
          call    clockoff        ;
          ld      hl,(outbuffer)  ;point to output buffer

          ld      a,(cyear)
          ld      (hl),a
          inc     hl
          ld      a,(cmonth)
          ld      (hl),a
          inc     hl
          ld      a,(cdate)
          ld      (hl),a
          inc     hl
          ld      a,(chour)
          ld      (hl),a          ;save the hours
          inc     hl              ;
          ld      a,(cmin)        ;
          ld      (hl),a          ;the minutes
          inc     hl              ;
          ld      a,(csec)        ;
          ld      e,(hl)          ;get the old seconds
                                  ;value for ZSDOS and the
          ld      (hl),a          ;seconds (all in BCD)

          ld      a,1
          ret                     ;and go away!
;
noset:    call    clockon
          ld      hl,(outbuffer)
          ld      a,(hl)
          ld      (cyear),a
          inc     hl
          ld      a,(hl)
          ld      (cmonth),a
          inc     hl
          ld      a,(hl)
          ld      (cdate),a
          inc     hl
          ld      de,buffer + 3
          ld      bc,3
          ldir                    ;YYMMDD -> MMDDYY
          ld      hl,0f8c0h       ;ASCII clock data address
          ld      de,buffer       ;
          ld      b,6             ;Six fields to convert
putloop:
          ld      a,(de)          ;
          and     0f0h            ;Strip off lo-nybble
          rra                     ;shift over to lo-part
          rra
          rra
          rra
          add     a,'0'           ;add in ASCII BIAS
          ld      (hl),a
          inc     hl
          ld      a,(de)          ;Get next digit
          and     0fh             ;Strip off hi-nybble
          add     a,'0'           ;Make into ASCII
          ld      (hl),a          ;and save it
          inc     de
          inc     hl              ;skip to next
          inc     hl              ;Skip delimiter
          djnz    putloop

          call    clockoff        ;
          ld      a,1             ;Flag that clock was set
          ret                     ;
;--------------------
clockoff: ld      a,(0ED13h)      ;Get F800's original
                                  ;map back
          ld      b,0f8h          ;
          ld      d,a             ;Point back to
                                  ;CP/M-86's bitmap
          ld      c,16h           ;RAM address map port
          out     (c),d           ;
          ei
          ret                     ;
;--------------------
clockon:  ld      (outbuffer),de  ;
          ld      b,0f8h          ;
          ld      d,05h           ;make 0f800 map to 2:8000
          ld      c,16h           ;RAM address map port
          di                      ;ACK!  NO Interrupts!
          out     (c),d           ;
          ret                     ;
;--------------------
buffer:                           ;Note -- this is the
cmonth:   db      0               ;format CP/M-86, has the
cdate:    db      0               ;date in -- we need to
cyear:    db      0               ;swap it around to what
chour:    db      0               ;ZSDOS wants.
cmin:     db      0
```

```
csec:       db      0

outbuffer:  ds      2

            .dephase
chsize      equ     $ - mywboot     ;Number of bytes to move.

            end
```

tually I do a function 0 and reboot—CP/M-86 doesn't notice anything weird is going on behind its back. It's for that reason that I think this could be done with little effort on other CP/M-like systems, such as TurboDos. They will see a program load, and make a bunch of BIOS and BDOS calls and then at some point end.

From the user's perspective, though, the system looks and behaves just like a normal system, except for some of the CP/M-86 features, like interrupt driven console input.

Occasionally I do have to exit Z-System (some things that fiddle with CP/M-86 are not accessible yet from my Z-System), so I have a simple alias to do this:

```
EXIT    poke 100 c3 00 ee;go
```

This forces a JP to the WBOOT routine in the emulator, which returns me gracefully to CP/M-86.

As for incarnations under MS-DOS: it seems possible. The trick is dealing with the differences in the structure of the disk. I am convinced that for now the best way to do this is to use a BDOS like Z80DOS, but strip out all the disk I/O calls, and change them to use the emulator's BDOS. This will allow for programs that hook into the character I/O functions (BYE, ZEX, AT and others) and should allow most programs to run fine using the emulator's BDOS for disk I/O. I will confess that I can't predict the behavior of a program like DU3 under such a system.

Note that 22NICE, a CP/M emulator for MSDOS (which I am currently analyzing to see if it will suit my odd purposes) allows for 'drive/user' areas to be mapped onto MSDOS directories. Setting up a named directory register to match the DOS names with the simulated user areas should make navigation compatible between the two systems.

At a later date, it would be interesting to use a full ZSDOS and DOSDISK—in theory, that should work quite well. But for now, I'm still in pursuit of a good fast Z80 emulator for my very slow clone.

Drop me a note at one of the addresses shown at the start of this article if you have a Vector4 and would like a copy of the operating system we've discussed here, with Z80DOS instead of ZSDOS.●

Computer Corner, from page 48

text you least expect it.

They supply plenty of documents to explain how their stuff works, but I am afraid it is mostly done by programmers and as such leaves the novice user in complete confusion. I finally traced my problems to one sentence that explained their MAKE utility. The utility has a tree structure starting from a single branch(a reference file) and you must carefully select that branch to get it to work correctly at all (must be only one file). That explanation is mine, theirs made less sense.

In the case of the PVCS program I can see some reasons for keeping some the installation and security secrets a bit cryptic. After all you do not want your user to know exactly how the security system works, else they will break it just for fun. That is why the course is most helpful, you get to see and play with setting it all up under controlled conditions (without others seeing you make the changes). That also was my biggest complaint about the course.

The objective was to learn how to set the system up under a LAN based environment, using some of the LAN features (it will work as single user as well). We learned on a non-LAN system and as such missed some of the extra playing that might be helpful later. It worked ok, but not as well as the real thing. It also prevented the instructor from doing some demos that a more complex LAN environment would have allowed.

LANs in the Training Lab

That showed me what a modern day instruction environment should be like. The instructor travels between several of the companies locations and as such really needs to take a system with him or her. Thanks to Motorola it will be possible to do a better job soon. How so?

In the next year or so it will be possible to hook a small transmitter/receiver to your serial or parallel port that connects to a LAN without any wires. For our traveling instructor it means a portable 386 with large hard disk and 6 or 8 of the LAN transceivers. Now all that is necessary is to have as many computers rented for the class as needed (at the training site) and the instructor carries everything else with them. Even what they carry is not so much really, the standard luggable 386 seems to be pretty standard carry on luggage these days. I saw at least 4 four people dragging them around in the airport in the first 5 minutes I was there.

For those teaching computer training programs, I feel the future will only get better. Think about being able to setup all your training and sample programs and directories once. Gone will be the days of redoing all the hard disks after each class and making sure you got all the floppies back from students. Sounds pretty good to me.

Brief but Sweet

Overall I found PVCS to be an ok product and definitely recommend you use some form of version control on your products. The thing to remember is that VCS still does not replace good and plentiful comments in the source file, it just makes it possible to find out who didn't comment their changes.

I will leave you with a hint about next time. FORTH day is just around the corner and WESCON will be happening as well. So next time...keep plugging away.●

> Basic research is when I'm doing what I don't know I'm doing.
>
> —Wernher von Braun

Real Computing

Minix Miscellany, Being Two Places at Once, and Hungarian Ghoulish

By Rick Rodman

Minix Miscellany

Before I got my 32381 chip, I was restless to do some programming on my PC-532. Since *gcc* requires the 32381, I was limited to assembly language. But I've noticed that assembly language under Minix is something nobody talks about. The listing shown *(Listing 1)* is an assembly-language program called *this.s* which makes two system calls under Minix. It's written in NS32 assembler. An 8086 version should be very similar.

You have to remember that Minix is a message-passing operating system, and the familiar system calls such as *write, read, ioctl,* et cetera, under Minix are actually implemented using message structures passed to the *real* system calls, which are *send* and *receive.* Constructing these messages in C is simplified by a number of macros, but under assembly you're on your own.

Each message can be 28 bytes or more in length. Each of the various system call messages has its own message structure. Some system calls need to get sent to FS, and some to MM. If you send a message to the wrong kernel task, it'll get some kind of error in reply.

In the example in *listing* 1, I build a "format 1" message for the *write* system call to send a message to the screen (actually, to *stdout*). Six fields in the message structure have to be set up, then two registers to identify the destination and type of service call, before executing the *svc* instruction (a far call under Minix).

In order to simplify matters, I just used the C startup routines and named my entry point *_main.* From looking at the listing, you'll see that setting up the message structure is quite a bit of work. Remember too, that for each system call, you need to know the message format, which field goes where, and what destination task to send the message to. Obviously, Minix has not been designed for convenience of assembly language programming!

I recently ported Richard Campbell's Tiny Basic to Bare Metal and hope to port it to Minix soon. Another Basic interpreter, written in C, is supposed to be available, but I have not been able to obtain source to it.

But now my 32381 has arrived and is installed and working, in all its resplendent numericity, in my PC-532. You have to admit, National has some high-tech-looking chips—with those bypass capacitors mounted on top, they really look like they mean business. And now, I'm all set for some heavy number crunching.

Only technoids look at chips anyhow, right? Intel's chips look like big blank walls, on which they have lately started to print fairly gaudy advertising. Intel's recent advertising blitzes "The Computer Inside" and "Intel Inside" are so moron-targeted that they're alienating anyone with even a passing awareness of technology. You'd think they were selling light beer or something.

386 Minix

In unrelated developments, patches have been developed for a 386 version of Minix, under which the Gnu tools are used for software development. With all the "smallness" of Minix out of the way, it becomes reasonable to implement subsystems like TCP/IP and X Window under Minix. However, Andy Tanenbaum, the author of Minix, originally intended it as a tutorial operating system for classes, and he doesn't want tremendous increases in the complexity of the package. For this reason, it is unlikely that the 386 version will be available from Prentice-Hall. Instead, if you want to run 386 Minix, you will have to either obtain ftp access to Internet and spend hours transferring files so you can have the fun of trying to cobble the system together from a bunch of patches, after which you can have the joy of trying to debug it. Some call the result "Advanced Minix".

See *Listing* 2 for the files you would need.

The Mars Hotel BBS in Maryland (1-301-277-9408) is supposed to have all these files. It also has a great deal of the /it/comp.os.minix postings. For the moment, I've decided against bringing up 386 Minix, because I already have Minix with GCC on my PC-532.

Now for some other Minix miscellany. The TTY (console, and external terminals—Minix works great as a multiuser OS on a PC) driver doesn't handle non-blocking I/O. It'll work fine for files and named pipes, but the TTY driver just doesn't have any code to implement it. If you want to write a program that only "samples" the keyboard, such as a screen editor with intelligent repaint or maybe a video game, you'll run into a problem.

Next, Andy Tanenbaum himself has distributed a new

Rick Rodman works and plays with computers because he sees that they are the world's greatest machine, appliance, canvas and plaything. He has programmed micros, minis and mainframes and loved them all. In his basement full of aluminum boxes, wire-wrap boards, cables running here and there, and a few recognizable computers, he is somewhere between Leonardo da Vinci and Dr. Frankenstein. Rick can be reached via Usenet at uunet!virtech!rickr or via 1200 bps modem at 703-330-9049.

Listing 2

```
 12259 Dec 12 1991 pub/Minix/oz/mx386.tute.Z John Nall's tutorial
  3623 Jul 11 1990 pub/Minix/oz/mx386_1.1.01.Z  patch to mx386
 45155 Jun 15 1990 pub/Minix/oz/mx386_1.1.t.Z   tar file with 386 patches
 12063 Jun 14 1990 pub/Minix/oz/bcc.tar.Ztar file with 386 C compiler frontend.
121962 Jun 15 1990 pub/Minix/oz/bccbin16.tar.Z  C compiler 16-bit binary
118254 Jun 15 1990 pub/Minix/oz/bccbin32.tar.Z  C compiler 32-bit binary
 43492 Jun 15 1990 pub/Minix/oz/bcclib.tar.Z 386 library sources.
 96151 Aug 14 1991 pub/Minix/oz/cpp.tar.Z C preprocessor (optional?)
 30659 Nov 15 1991 pub/Minix/oz/cppmake.tar.Z   Earl Chew's cppmake program.
```

version of *cdiff* which uses CRCs to ensure that the version being patched is the right one, and that the result of the patch is correct.

Computer-security fanatics who were enthralled by the account in *The Cuckoo's Egg* of breaking the password file of a Unix computer may be interested in a surprisingly simple program which implements the concept. It works under Minix just as well as under Unix. The general idea is that most users use either proper names or dictionary words as their passwords. Since the encryption algorithm is accessible, it's quite easy to run through a dictionary, encrypting words and comparing the result to the encrypted password in the password file. No matter what system administrators tell people, users keep using dictionary passwords. (Not me, Bub! I use made-up but pronounceable words like 'zlofsnerb').

Last time I discussed the Named Pipes under Minix. Alas, there are problems. It seems that each writer receives his own file pointer, so that multiple messages sent at the same time will cause trashing of messages. The way this is fixed in Unix is by using two bytes of the "i-node" for a write pointer; patches have been developed implementing this solution, but not everyone is very happy with it.

The above miscellaneous items are available on my BBS (1-703-330-9049, evenings, up to 9600 baud) in the form of two big text files, which are simply transcripts of Usenet messages, called uminix.21 and uminix.22. If you want only a brief item and don't want to transfer these masses, leave me a message.

Being Two Places at Once Has Its Drawbacks

I know *TCJ* is not a Windows-related magazine, but many of you may have fooled around with Windows, and if you have any occasion to do programming under it, you need to know what I am about to pass on to you.

In a standard state-machine design, states are *atomic*—each state, or message, executes to completion before another is processed. You can rely on this condition to save yourself a lot of worry about data consistency. Multiple state machines can be synchronized by semaphores or other operating system mechanisms.

I've learned to my chagrin that, under Windows, you can be two places at once and can't do anything about it.

The problem is that Windows is not preemptively multitasking. When doing something time-consuming, you have to voluntarily relinquish the processor. The usual way is to call a function named *YieldProc* or something like that, which has a quick message loop allowing other messages to be dispatched—but guess what? Messages can be dispatched to *your* program as well! For example, you might be computing values to be displayed in a window, and the window repainting message comes in to display them before you're finished!

Unfortunately, because Windows is not a real OS, there's not much you can do about it. There are no semaphores or other mechanisms. Many commercial libraries include *YieldProc*-like calls without telling you, too. The upshot is, you have to program extremely defensively. You can never know, in a message-processing routine, whether several different routines in your code are being executed at once.

The most common problem is with *WM_PAINT*. Make sure that your *WM_PAINT*, *WM_SIZE* and other window-painting-related routines do little or no computation, so they can execute atomically. I know it's tempting to recalculate on display, but don't—keep the data in a buffer. With

See Real Computing, page 29

LISTING 1 **ASSEMBLY LANGUAGE UNDER MINIX**

```
;test assembly language program
;Assemble with: cc this.s -o this
;This causes the crtso.o file to be linked first.

.static
mytext:
.byte   'This is a test',13,10,0
;               12345678901234

;Minix message structure will be built in this buffer

message:
.blkd   7

;Offsets for message structure. Each system call has a
particular
;message structure; some need to go to FS, some to MM.

off_src:        .equ    0
off_type:       .equ    4
off_mli1:       .equ    8
off_mli2:       .equ    12
off_mli3:       .equ    16
off_mlcpl:      .equ    20

.program
;start::
_main::          ;Entry point from crtso.o
movd    message,r1
movd    0,off_src(r1)   ;build message format 1 for write
movd    4,off_type(r1)  ;SC 4 = write
movd    1,off_mli1(r1)  ;1 = stdout
movd    16,off_mli2(r1) ;count of bytes
movqd   0,off_mli3(r1)  ;unused
movd    mytext,off_mlcpl(r1)    ;buffer pointer
movqd   1,r2            ;destination = FS
movqd   3,r3            ;send and receive
svc                     ;make the call

;return result is in message.off_type

;now need to exit. (actually don't; since crtso.o was
linked first,
;returning is all that's necessary.

movd    message,r1
movqd   0,off_src(r1)
movd    1,off_type(r1)  ;SC 1 = EXIT
movqd   0,off_mli1(r1)  ;return 0
movqd   0,r2            ;destination = MM
movqd   1,r3            ;send only
svc                     ;make the call

ret     0               ;should not get here
```

Interrupts and the Z80

By David Goodenough

Most of the time when a CPU is executing a program, it will get on with the job with almost single minded devotion to the task. It has the program to execute, and it sits there executing instruction after instruction. Unfortunately the real world is a little different: there are sometimes external stimuli that need to be dealt with. For example, what does the system do when you press a key on the keyboard?

The naive approach to this would be to only inspect the keyboard when you're actually waiting for input, a technique known as polling. This has the advantage that it's very simple to implement, but has the obvious disadvantage that keys pressed when the keyboard isn't being checked stand a very good chance of being lost.

Wouldn't it be better if we could temporarily stop the CPU running the program and go off and handle a keyboard press, whenever one happens. This would let us save all the keystrokes in a buffer somewhere until they're needed. This is what an interrupt does: it interrupts the CPU's current task, and makes it go off and do something else, and when this second job is finished the main task can be continued, as if nothing had ever happened.

The Z80 is an interesting CPU in it's handling of interrupts. It allows the programmer to select between three different modes, which can be selected under software control. The third of these modes is very powerful, working at a level that would be expected more in a mini-computer rather than an eight bit micro.

The first of the three modes, mode zero, was designed to be the same as the interrupt handling on the 8080. Since the Z80 is an upgrade of the Intel 8080, that was a logical decision to make. In this mode, when the 8080 (or Z80) is acknowledging an interrupt, the interrupting device is able to write a single instruction onto the data bus that the CPU will then execute. In almost all cases this will be one of the single byte call instructions that the 8080 and Z80 support. The net result of this is that when the interrupt occurs, the CPU calls to a specific location in memory, and is able to execute an interrupt routine residing there.

The second mode, mode one, is the simplest, in that all it does is a unilateral call to a specific location, so that the code to handle all interrupts is in one big subroutine. As the address that the CPU calls to cannot be changed, this is the least versatile, but is the easiest to work with in a simple application.

The third mode, mode two, is where the Z80 stands head and shoulders above the rest. This is a complex method, but once it's learned, the power it gives can be put to good use. The Z80 has a special eight bit interrupt register, which can be loaded from the accumulator. This register is intended to hold the address of a 256 byte page somewhere in the address space of the Z80. Taking my Televideo system here as an example, the interrupt register contains 0F2H, so the page being addressed is 0F200H through 0F2FFH. When an interrupt is generated, in a similar manner to mode zero, the interrupting device is expected to write a byte onto the data bus. This byte becomes the low eight bits of the address in the interrupt page. This allows selection of any one of 128 sixteen bit words in the page. For example, when the clock tick interrupt goes off, the CTC chip puts 068H on the data bus, and the Z80 then builds the address 0F268H. Then the Z80 reads the word at that address, and calls to that location.

Figure 1

```
#dwf200                 ; dump words, from 0F200H

F200: F1C3 C3FC F9BA 80C3 C3F2 F2A6 BEC3 C3F2   ................
F210: F334 2BC3 C3F3 F323 BDC3 C3F4 F4A9 BFC3   4..+..#.........
F220: C3F4 F4C4 C9C3 C3F4 F534 59C3 C3F5 F299   ........4..Y....
F230: CEC3 C3F4 F28A 8EC3 35F2 86FC 2513 2620   .........7...% &
F240: 9106 0019 8000 E500 0001 0000 0000 FFFF   ................
-------------------------------------------------------------------

F250: F3F6 F3F6 F3F6 F3F6 F3F6 F3F6 F428 F467   ............(.g.
F260: F3F6 F3F6 F3F6 F3F6 F33D F3F6 F3F6 F3F6   ........=.......
                          ^^^^
F270: F3F6 F3F6 F3F6 F3F6 F3F6 F3F6 F3F6 F3F6   ................
```

Figure 1 shows a dump of the page in question shows. [Note—Most of the vectors point to a "null" subroutine that simple re-enables interrupts and then returns.]

Let's look then at the code at 0F33DH:

```
#1f33d
F33D  LD   (FCC2),SP
F341  LD   SP,FCC2
F344  PUSH AF
F345  PUSH HL
F346  PUSH DE
F347  LD   HL,(F239)
```

David Goodenough programs both professionally and for a hobby. Professionally he uses C for projects under both DOS and UNIX. In the hobby world, his first choice is Z80 assembler. He cut his teeth in the computer world programming a National Semiconductor SC/MP in HEX in 1974, and has been at it ever since. He lives at 1236 15th Avenue, San Francisco, CA 94122, and can be reached at (415) 665 3721 when his computers allow him to use the phone.

```
F34A  INC  HL
F34B  LD   (F239),HL
...
```

When the Z80 reads from 0F268H, it reads the word 0F33DH, as shown above, and starts there. As the disassembly at 0F33DH shows, this is the entry of the timer tick routine. This means is that the Z80 can have 128 separate interrupt routines accessible, and each interrupting device can select what routine to use by providing the low word of the address in the interrupt page.

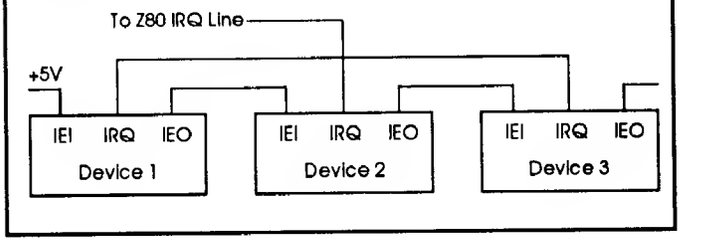
This explains how the Z80 handles interrupts as far as the software is concerned, but what happens at the hardware level?

The Z80 itself has only one general purpose interrupt request line, but there may be several devices that wish to generate interrupts. How do they arbitrate who has the CPU's attention?

Note that the CPU actually has two interrupt lines, IRQ (Interrupt request, which is what is under discussion here) and NMI (Non-maskable interrupt). NMI is generally reserved for very special use, since it cannot be disabled. It operates in a manner similar to mode one, calling to a specific address. Typical uses for NMI include fatal error handling, when some error condition absolutely *has* to be dealt with.

Figure 2 shows (rather crudely) what happens. If device 1 does not need to interrupt, it lets it's IEO (Interrupt Enable Out) line follow it's IEI (Interrupt Enable In), so a +5 comes out, and device 2 also sees +5 on IEI and (being idle as well) passes +5 out of it's IEO to the next device in the chain. This is the "idle" condition: any device that sees +5 on it's IEI line is at liberty to generate an interrupt.

Figure 2



When device 1 wants to interrupt, it asserts IRQ to inform the Z80 that it wants attention. All the IRQ lines from the interrupting devices are gated together to go into the Z80's IRQ line. It also drops it's IEO to 0v, and this condition propagates down the line, device 2 sees 0v on it's IEI and drops it's IEO to 0v, and so on. Any device that has 0v on it's IEI line can't generate an interrupt, so device 1 knows that no other device in the chain can interrupt.

Similarly, if device 2 wanted to interrupt, and device 1 was idle, device 2 would have +5 on it's IEI, and it would drop it's IEO to 0v. This has the same effect: device 2 knows that no device further down the chain can interrupt because the 0v it has on it's IEO is disabling them, and it also knows that devices above it in the chain (device 1, in this case) won't want to interrupt because they see +5 on their IEI line. So again, device 2 knows that it has the exclusive attention of the Z80.

Listing 1

```
icode:  di                      ; make sure further interrupts are disabled
        push    hl
        push    af              ; save just two registers
iclp:   in      a,(0x2d)        ; read the status port
        add     a,a             ; check the MS bit
        jr      nc,icdone       ; MS bit not set, so it's time to exit
_here_: ld      hl,0            ; filled in later [*] - this will point
                                ; hl at the buffer area
        inc     (hl)            ; increase the save index
        res     5,(hl)          ; but limit it to 32 bytes
        ld      a,(hl)          ; now get the updated index
        inc     hl              ; point hl at the buffer
        add     a,l
        ld      l,a
        jr      nc,hok
        inc     h               ; these last four instructions add a to hl
                                ; this effectively points hl at the correct
                                ; place in the buffer to save the character
hok:    in      a,(0x2f)        ; read a character from the modem port
        ld      (hl),a          ; save it in the buffer
        jr      iclp            ; jump back to try again
icdone: pop     af              ; all done: restore the registers
        pop     hl
        ei                      ; re-enable interrupts
        reti                    ; and return from the interrupt
endi:
```

Listing 2.

```
install:
        di                      ; turn off interrupts while we do this
        ld      hl,(0xff00)     ; the word at 0xff00 contains the address
                                ; of the buffer we'll use.
        xor     a
        ld      (hl),a          ; set the first buffer index to zero
        ld      (_base_),hl     ; save the base address of the buffer area
        inc     hl
        ld      (hl),a          ; set the second index to zero
        ld      (_here_ + 1),hl ; _THIS_ is where the address in the interrupt
                                ; code gets filled in: we point it at the
                                ; second index
        ld      de,33           ; move a further 33 bytes up. This allocates
                                ; 34 bytes altogether with the inc hl above:
                                ; two for the indexes, and 32 for the buffer
                                ; itself
        add     hl,de
        ex      de,hl           ; get that address to de for the target
        push    de              ; and save a copy on the stack
        ld      hl,icode        ; source is icode
        ld      bc,(endi - icode)
                                ; get the length of the interrupt routine
        ldir                    ; go move it
        ld      c,0x10          ; This is a byte value that will be used to
                                ; turn on interrupts in the UART
        pop     de              ; get the interrupt address back
        jr      endit
```

Listing 3.

```
do_exit:
        di                      ; interrupts off while we work
        ld      de,(ivec)       ; ivec contains the original vector,
                                ; when the code below is explained, it will
                                ; become obvious how it got there
        ld      c,0             ; set c to zero to stop the UART generating
                                ; interrupts

; OK, this is the code that actually does the work.
endit:  ld      a,i             ; fetch the refresh register - this is the
                                ; page address of where the vectors live
        ld      h,a             ; save it in h
        ld      l,0x78          ; we want the vector at XX78h
        ld      a,(hl)          ; get the low byte of the old one to a
        ld      (hl),e          ; install the low byte of the new one from e
        ld      e,a             ; save the old low byte in e
        inc     hl              ; point to the high byte
        ld      a,(hl)
        ld      (hl),d
        ld      d,a             ; repeat the switch, but this time with d

; OK, those last few instructions swapped the word in de with the word
; in the interrupt page: this means we have the new vector installed
; from de, and the old one saved back in de
        ld      (ivec),de       ; and we save the old one away in ivec.
        ld      a,0x65
        out     (0x28),a
        ld      a,c
        out     (0x20),a        ; initialise the UART with the value in C
        out     (0x27),a
        ld      a,0x64
        out     (0x28),a        ; and a little more setup,
        ei                      ; interrupts back on, now that we're done
        ret
```

The Z80 will acknowledge the interrupt, and at that time device 2 will place the correct byte on the data bus that the Z80 uses to select the interrupt routine, and device 2's interrupt will be serviced.

So far, I've talked about how interrupts work, and what different methods exist for dealing with them at a software level. Now let's look at a real life example. As many people can attest, terminal programs written for Z80 machines sometimes exhibit a very bad habit of losing the first few characters of each line. This is such a well known problem that many BBS's in the CP/M and Z-System community offer the ability to transmit some NUL characters at the start of each line, to act as "sacrificial" characters that can be lost without problems.

However this is really only curing the symptom, to address this problem it is necessary to find out why the characters are being lost. Taking the Kaypro machines as an example of this, when the terminal program outputs a linefeed to the screen, this will usually cause the screen to scroll. On the Kaypro, this means that the Z80 has got to shift almost 2K of data, using a LDIR instruction. On a 2.5MHz system this takes quite a bit of time, in fact it takes so much time that while the scrolling is in progress, there is sufficient time for several characters to arrive from the modem, and these are the ones that get lost. What we are seeing is a polled system (the terminal program polls the modem port), and there are times when the CPU is so tied up doing something that it can't get to the modem fast enough.

Suppose we were able to use an interrupt for this, so that whenever there was a character waiting at the modem port, an interrupt would be generated. Now what happens? The linefeed is processed, and the CPU goes off and starts moving characters on the screen. While it's doing this, a character arrives from the modem, causing the CPU to stop shifting screen characters for a few moments, just long enough to get the character from the modem and save it somewhere. Then it carries on shifting screen characters. In the time taken to move the entire 2K of data needed, it might get interrupted five or six times, or even more, but none of the characters will be lost because as each one of them arrives, the CPU saves them away. What is the result of doing all of this? No more character loss.

Another issue that needs to be addressed is what pitfalls exist in using interrupts. If the CPU is in the middle of doing something, and then all of a sudden it has to drop everything to go do something else, it is absolutely essential that everything is dropped in such a manner that it can be picked up again.

Taking another look at the timer interrupt routine, the first thing it does is:

```
F33D  LD   (FCC2),SP
F341  LD   SP,FCC2
F344  PUSH AF
F345  PUSH HL
F346  PUSH DE
F347  LD   HL,(F239)
F34A  INC  HL
F34B  LD   (F239),HL
```

It starts by saving the stack pointer, and then loading the stack pointer with the address of an area of memory that is

Listing 4.

```
; modist - return with z flag clear iff there is a char waiting at modem port
modist:
        ld      hl,(_base_)     ; go get the address of the buffer area
                                ; where things are saved
        ld      a,(hl)          ; fetch the first index
        inc     hl              ; point at the second on
        xor     (hl)            ; set the accumulator and z flags based
                                ; on whether there's data in the buffer
ret

; modin - read char from modem port: modist has been used to check it's there
modin:
        ld      hl,(_base_)     ; get the buffer address again
        inc     (hl)            ; increase the read index
        res     5,(hl)          ; but limit it to 32
        ld      e,(hl)          ; fetch the offset to e
        inc     hl
        inc     hl              ; point hl at the buffer
        ld      d,0             ; extend e to de
        add     hl,de           ; point at the byte to read
        ld      a,(hl)          ; and go get it.
        ret
```

set aside for use of this interrupt routine alone. The very next thing it does is to push several registers onto the stack. This is what it has to do to preserve the state of the CPU before the interrupt happened. After pushing the registers it gets down to the real work, starting by increasing a tick value at 0F239H. It goes without saying that since AF, HL and DE are the only registers saved (BC and the index registers aren't), these are the only registers that can be used. If it became necessary to use BC in this interrupt routine, then the entry code of the routine would have to push BC onto the stack, so that the original value could be restored on exit.

This typifies the main thing necessary in writing interrupt routines it is essential to restore the CPU state to exactly what it was on entry to the interrupt routine. Changing so much as one bit in a register, or the state of one flag, will eventually cause a problem, and problems like that are almost impossible to catch, since they will not be easily repeatable. An interesting side note to this is that the Z80 has another nice feature that was designed mainly for use in interrupt handlers. It has two sets of registers: the first set that can be accessed with instructions like 'ld a,(01234H)' and 'add hl,de'. However there is a second set that are normally hidden away, but these can be exchanged with the main set very quickly. The idea here is to use only the first set of registers in the main line code, and reserve the second set for interrupt handling. This allows a very fast save and restore, which in turn is a real advantage to people writing interrupt routines. Sadly, in real life, code often uses these alternate registers, hence making them unavailable for use in interrupts, although in a dedicated environment where the programmer has complete control over the system, this could be done.

It is unfortunate that interrupts are always very specific to the machine being used, since the examples provided will only be actually testable in it's current form by people who have Televideo 803's or TPC-1's. To show a complete interrupt setup, including installation of the code, the interrupt code itself, and how it is removed when the program exits, I am going to pull some code from QTERM. First of all, the interrupt routine itself. See *Listing 1*.

OK, that's the interrupt routine itself. One thing that should be fairly apparent is that it is written to be as small and fast as possible: this is always a desire in interrupt routines: make them as quick as possible, so that they steal the least amount of time from the main task.

Another interesting point is that the buffer address is not a constant. The reason for this is that the interrupt routine gets installed up in the BIOS in a small buffer up there, this is necessary because the Televideo 803 can switch between several pages of memory, and there is no way of guaranteeing that page 1 (which is where CP/M programs normally reside) will be paged in when an interrupt occurs. So it is necessary to place the interrupt routine in high memory, above the area where the paging takes place. How this is all done is shown in the installation code. See *Listing 2*.

Well, it looks like I kinda left that hanging, however when QTERM finishes it's necessary to re-install the original vector. If I didn't, then the next time a modem interrupt occurred it would call to the buffer area. Since this is used by other programs, there would be no guarantee that our interrupt handler would still be in place, and so the CPU could wind up executing anything. So what does the "de-installer" do? Look at *Listing 3*.

The important things to remember from the above are that while the vector is being changed in the interrupt page, it is *essential* to have interrupts disabled. During the update process, there is a small window when the integrity of the table is not complete. If an interrupt arrived during that window, it would almost certainly cause a system crash. Also note how the original vector is restored on exit, and how the UART has interrupts disabled on exit. This sort of cleanup is also very important to prevent problems after QTERM has exited.

Finally a couple of small routines are needed to read characters back from the buffer where the interrupt routine has saved them. We have them in *Listing 4*.

In conclusion, interrupts provide a means for external "real world" events and stimuli to be handled by the CPU without the necessity for continuously polling. This can have a dramatic improvement on system performance, both in terms of the CPU cycles saved by not needing to poll, and because the event will often get much more rapid processing. Of course "There ain't no such thing as a free lunch": it is almost always far more complex to produce an interrupt driven system. Despite of this, the advantages of an interrupt system generally outweigh the disadvantages. While it can take a while to get comfortable writing interrupt drivers, it is well worth taking the time to do so, and learning by doing is usually a very good teaching help. From personal experience, I can assure you that you have a very good chance of locking up your machine several times before getting it all sorted out, but it pays back many times over when all the bugs are out and the system finally works.●

Real Computing, from page 25

WM_COMMAND, there's not much you can do. Code with the assumption that it will be re-entered, possibly multiple times.

Think of Windows as being like sausage. It looks great, tastes great—but you don't want to know what's inside it. OS/2, by contrast, is a real OS, and needs none of this *YieldProc* nonsense.

Hungarian Ghoulish, or What a Pane

Not meaning any slight to our friends of any national persuasion, I still find that so-called "Hungarian notation" Microsoft seeks to push on everyone offensive. My C experience of ten years teaches me to use upper-case letters to denote macros. At a glance, I can tell whether a data item is a macro or the "real thing." This crazy new style makes your programs look like some kind of advertising. They got it from Pascal—the language where you can't even tell if *SillyNonsense* is a variable or a procedure, for crying out loud. If you ask me, It'sALotHarderToRead, and I'm going to keep using "break characters." That way, the Windows and OS/2 calls stick out like the sore_thumbs_they_are.

Next time

Next time I hope to return to the home-control topic with some neat new X-10 hardware and ideas. Also, I hope to fiddle around with some image compression and decompression on my PC-532. Just because there are parts out there called DSPs doesn't mean you have to have one to do it!●

8 MHz on an Ampro

Double Your Pleasure, Double Your Fun. Double Your Clock Speed and Watch It Run!

By George Warner

A few years back I was browsing through a local BBS when I read a message telling how to modify an Ampro Little Board Z80 to run at 8 Mhz. My silicon-slave of the time was a very modified California Computer System S100 machine happily chugging along at 4 Mhz Always interested in enhancing my system's performance via hardware hacking, I immediately made a rush for my bookshelf of hardware manuals; 23 mods and 1209600 wait states later (okay, two weeks), my CCS was okay. For eight! It was almost six months later before I ran across additional dialog concerning the Ampro 8 Mhz mod. Returning to the original message my curiosity was peeked by the extensive number of modifications (six cuts, eight jumpers and 120 ns DRAMs) required to "make it work" (with wait states!). My Ego having been well stoked with my previous success, I mentioned to several fellow BBS'ers my opinion of this supposedly "simple" 8 Mhz mod. Being the extremely practical persons they are, they immediately capitalized on my comments and suggested I "put my money where my mouth was". A very generous individual donated an unused 1A to the cause and "the game was afoot!" Had I known that he would later be taking over editor-ship of this publication my brain might have kicked in and a hastily retreat taken. But noooooo. 'nough said. Now for the nuts and bolts.

A five-minute addition to my normal daily commute dropped me into Ampro's lobby where a kind individual delivered six crisp 14"x11" schematics for the 1A and 1B. The first task at hand was to "speck" the standard parts in order to determine what parts just weren't fast enough and would have to be replaced.

Obvious (to me) was the Z80ACPU, CTC & DART (4 Mhz max.). The CPU was quickly swapped with it's 8 Mhz counterpart, the Z80H. Not so lucky with the DART, it wasn't available locally in any speed over 4 Mhz. Lucky for us the rumor mill said (pause) that the SIO-0 was pin-compatible. A quick call to Zilog verified this true and also provided some additional information. Most 6 Mhz SIO's were 8 Mhz manufacturing failures! If the synchronous test failed at 8 Mhz but passed at 6.5 Mhz the part was produced as a 6 Mhz part! Since the original DART we were replacing didn't run synchronous this was no great loss to us. I replaced the DART with a SIO-0B (6 Mhz). A quick look in the speck-books confirmed a lesson learned from my CCS system. The CTC could work with a 8 Mhz as long as it's clock inputs weren't faster than 4 Mhz. In other words, you can access it at 8 Mhz. It just can't count faster than 4 Mhz. No change there. The only things left to consider are the WD1770 floppy disk controller, the NEC5380 SCSI controller (1B only) and the DRAM. Both the WD1770 and NEC5380 will work at 8 Mhz. No problem there. That leaves the memory. We will get back to it later. For now....

We swap in the faster parts, power-up, boot and everything still works at 4 Mhz. Time for the knife. Note: My original mod was done on a little board 1A. The references in parentheses. below are for the 1B. They have been tested and do work.

First things first, I disconnect the Z80's clock input from 4 Mhz where it is generated on U10(U3)P13 and reconnect it to 8 Mhz where it is generated on pin 14. Note: this has also changed the CTC & DRAM address mux. delay flip-flop clock frequency to 8 Mhz. Not important now, but later.... With my finger crossed I then turned on power; hit reset and boom! nothing. Reset again and... nothing. Well no big surprise there, the DRAM just ain't' hacking it. Wait; the floppy drive has a light on! And the door is open! Duh, Hello? Insert boot disk, close door, reset and... Click, click, click (drive noise) and... nothing. Hmmm. Wait, the boot program is in EPROM! It works! But why did it stop? Well no big surprise there, the DRAM truly ain't' hacking it. Time to hit the books.

DRAM's are funny little Anti-mules; I don't think any two people (or manufacturers) will agree on which timing is the most critical. (The truth: the one that keeps it from working is the most critical.) In a nut-shell (where I belong) this is how DRAM read and write cycles are supposed to work:

1. The DRAM address lines are driven with the row address. Usually the CPU's A8-15.
2. RAS* is pulled active (low) by the CPU.
3. If this is a write cycle, the CPU puts data on the DRAM's data input.
4. If this is a write cycle, W* is pulled active (low) by the CPU.
5. The DRAM address lines are driven with the col. address. Usually the CPU's A0-7.
6. CAS* is pulled active (low) by the CPU.
7. If this is a read cycle, the DRAM drives it's data output.
8. If this is a write cycle, the CPU releases (high) W*.
9. The CPU releases (high) both RAS* and CAS*.

The (very debated) critical design criteria (in random order) for DRAM are:

1. Row address setup time.	(#1 to #2)
2. Row address hold time.	(#2 to #5)
3. Precharge or Recharge time.	(#9 to next cycle's #2)
4. Data valid to write enable time.	(#3 to #4)
5. Read access time.	(#6 to #7)
6. Refresh time.	(See text)

The row address setup time is the time from when the row address have settled on the DRAM address lines (#1 above) to when RAS* is pulled active (low) (#2). The 4116 DRAM specification gives this as 30 nsec. minimum.

The row address hold time is the time from when RAS* is pulled active (low) (#2) until the DRAM address lines can be changed (#5). The specification shows 25 nsec. minimum.

The precharge or recharge time is the time after one cycle (#9) that allows the DRAM to charge internal parts prior to the next cycle (#2). The minimum specification is 150 nsec.

The data valid to write enable time is the time from when the CPU data lines settle (#3) until the write line (W*) is pulled active (low) (#4). The specification for this is 40 nsec. minimum.

The read access time is measured from when the CAS* line goes active (low) (#6) until the DRAM's data output is stable (#7). Almost all DRAM chips SIP's & SIMM's are sold by this speed (and their density of course). Typical 4164's for the Ampro's I have seen range from 150 to 200 nsec. For a 8 Mhz clock, each cycle is 125 nsec. The Z80 has a worst case read cycle of 1 1/2 clock cycles for the M1 (op-code fetch) cycle. 125 nsec. * 1.5 = 187.5 nsec. This means that 200 ns. DRAM won't work at 8 Mhz As you will see later, because of chip delays 150 nsec. parts are a tight squeeze.

One of the most annoying differences between static and dynamic RAM's is the DRAM's refresh. If the contents of DRAM's aren't periodically refreshed it "forgets". The DRAM RAS* only refresh cycle consist of:

1. The DRAM address lines are driven with the refresh row address. This is usually incremented after every refresh access.
2. RAS* is pulled active (low) by the CPU.
3. RAS* is released.

This must happen once for each row of address every 2 msec. Since the 4164 has 128 rows, one row needs to be refresh every 16 usec. On the Z80 refreshes occur during the last two clock cycles of every M1 (op-code fetch) cycle. During this time the Z80 is involved in internal operations. Since a Z80 operating at 8 Mhz takes 3 usec. to execute the longest instruction in its set, M1 cycles occur frequently enough for the memory to be refreshed.

Time for the o'scope.

1. Row address setup time.	(30 nsec. min.)	45 nsec.
2. Row address hold time.	(25 nsec. min.)	85 nsec.
3. Precharge or Recharge time.	(150 nsec. min.)	100 nsec.*
4. Data valid to write enable time.	(40 nsec. min.)	125 nsec.
5. Read access time.	(150 ns. min.)	85 nsec.*
6. Refresh time.	(16 us. max.)	3 usec.

Well, worst thing worst, how can we extend the read access time to greater than 150 ns.? I guess we could insert wait states.... *Wrong! I want 8 MHz!* How about making CAS* active (low) sooner? According to our schematic, the first flip-flop of U18 (U37) is used to delay MREQ* from the Z80 for 125 nsec. to generate our ROW/COL address mux. on pin 5; the second flip-flop is then used to delay this by an additional 125 nsec. to generate CAS* on pin 9. Although not mentioned as critical, the 4164 DRAM's column address setup time is 30 nsec. The 125 nsec. delay introduced by the second flip-flop seems a little unnecessary. We do need some delay however. A quick trip to the 74S74 spec sheet shows it's clear (C*) to output low delay to be 30 nsec. Just what we need! By disconnecting the clear input on pin 13 from it's pull-up resistor and connecting it to the ROW/COL address mux. signal coming in on pin 12; we can extend our read access time by about 90 nsec. Some quick math reveals 85 + 90 = 175. Everything looks good on paper. So, let's put the proof into the pudding.

How do we disconnect pin 13 from its pull-up resistor? The first time I did this I desoldered U18(U37) and looked under it. A trace goes from pin 1 to pin 13 and then to U19(U35)P1. This signal needs to be cut and reconnected around pin 13. Once I determined what had to be done; I quickly came up with an easier way: Rather than removing U18(U37), cut off pin 13's lead where it hits the P.C. board, bend it up and then solder it to pin 12.

Quick hint: When you are working on a board which costs a hundred dollars and more and need to remove, replace or socket a fifty-cent IC, cut off all the leads to the IC with dikes; desolder the leads from the board one at a time and then solder in the new part or socket. I've seen experienced techs destroy two-hundred dollar boards trying to save a fifty cents part! Using this hint has removed all but a few of my needs for a de-soldering station.

Crossing my fingers again I turned on power; Insert boot disk, close door, reset and... Click, click, click (drive noise) and... up it comes! I guess the Precharge or Recharge time (#3 above) wasn't as critical as was thought. *Ta-Da!* 8Mhz!

Just in case you were bored by this exciting (yawn) discussion and skipped to the end, here is the good stuff:

1. Replace CPU with Z80H (Z0840008PSC).
2. Replace DART with 6 or 8 Mhz DART or SIO-0. (My 6 Mhz SIO-0 works fine.)
3. On the component side of the board cut the trace coming from U10(U3) pin 13 going to a small through pad. Connect this pad on the circuit side to U10(U3) pin 14.
4. On the component side of the board, cut the lead coming from U18(U37) pin 13 AT THE BOARD; then bend it up and over and solder it to pin 12.

That's it! I have lost count of the number of boards (1A's & 1B's) I have done this to and have had *no* problems. Joe Wright of Alpha Systems even came up with a quick patch for the BDOS/BIOS to determine the clock speed by using the CTC clocked at 2 Mhz as a reference.

```
SPEED:  LD    C,CTC0    ; Get port address of
                        ; CTC -> register C.
        LD    A,3
        OUT   (C),A     ; Reset CTC
        OUT   (C),A     ; Again.
        LD    A,047H    ; set mode & load divisor command.
        OUT   (C),A
        LD    A,100     ; output divisor
        OUT   (C),A
        IN    L,(C)     ; Get first value from counter
        IN    H,(C)     ; Get second value from  counter
                        ; (12 cpu cycles later)
        LD    A,H       ; Move second value -> Acc.
        SUB   L         ; Subtract first value
        CP    6         ; Compare to 6 counts.
        RET             ; 4Mhz = 6 counts (No carry)
                        ; 8Mhz = 3 counts (carry).
```

Note: I haven't been able to make this work reliability with Terry Hazen's (n/Systems) 1 Mbyte memory disk.●

TCJ The Computer Journal Market Place

Hardware Heaven

Dallas Smartwatch and Data Books

By Paul Chidley

What's Going On Here?

The last few years have seen many changes in Europe and the former USSR. On a personal note, it has seen me switch jobs, move west, and give up on Ohio Scientific 6502 for the land of Z-80, Z-systems, and other z-things. It has also led me to build the Yasbec and in the process meet many new friends. One glaring fact has come to my attention from talking to Yasbec builders and other such z-people. Even in the CP/M world there just aren't many hardware hackers left out there. (*TCJ* folk excluded, of course!)

Yes, I know, there are lots of engineers that have degrees up to their ya hoos, but have they ever seen a soldering iron? (*ya hoo:* a term shouted as loud as possible at hockey games when the Calgary Flames score.) The average tech today can solder fine but if the part isn't available from Radio Shack he's lost as to where to look for it. Or how about the average PC user that doesn't want to know what's under the hood.

If you disagree then prove me wrong, write an article for *TCJ* to tell the world (or our small part of it) what you are doing. You don't have to be an exceptional writer. This article proves that. You just have to write something resembling English about almost anything and *send it it!* The worst that can happen is that Chris prints it! [That's hardly the worst thing to happen. The worst is that no one will know all the things we all do and conclude that only plug–and–play users remain if you *don't* share your work.—Ed.]

So against my better judgment I am going to attempt to write a hardware hacker's column. You know; the chip-of-the-month club, what's new in old surplus junk, where to buy a Z8018012VSC, how to avoid the $200 minimum order, how to get samples and data books, etc.

There is nothing more frustrating than seeing a plea for help on the old BBS from a guy living in Cupertino or Mountain View (the valley) that just can't find *anywhere* to buy parts. They should repossess his couch! My old job sent me down there at least once a year and like most places it is a gold mine of parts. You just have to know where to look. I am seriously considering a family vacation driving to Disneyland just so I can go via "the valley" for a shopping trip. I'm sure my wife would understand. Would the bank?

Enough rambl'in so lets move on to the meat.

The DS1216E

I suggested that a Dallas Semiconductor DS1216E be installed under the EPROM in the Yasbec to provide a battery backed up real time clock. The number of questions about this one part amazed me. The DS1216 is a neat part, or more accurately module. Basically it is a socket. You simply remove the monitor ROM from your system and plug in the DS1216 in its place and place your ROM into the DS1216.

The Smartwatch is accessed by executing a very specific pattern on the address lines. When you complete the right combination, the DS1216 disables the ROM plugged into it and instead makes itself available on the data bus. No cuts or jumpers to your circuit; just plug it in. If you see an IBM type ad for a "Smartwatch" or "Smartclock," chances are it is a DS1216E. As far as I know there is only one similar product on the market. But don't buy a DS1216E from an ad for clones. They usually come with software to use it on an IBM at twice the price of the chip alone.

How can you tell if it is a DS1216E? The IC in the middle of the socket will have DS1216 printed on it. But where's the "E?" Look again, the socket comes in various flavours (B-F) and the magic letter is etched in the copper of the circuit board at the end of the IC. It is usually about twice the height of the IC. Its so big that no one can see it.

Programs to talk to the SmartWatch can be found on most CP/M boards, GEnie, Z-Nodes, CRS, et cetera.

I won't go into the details of how this chip works. That's what data sheets are for. You have a Dallas Semiconductor Data book right? Why not?

Data Books

Engineers would be lost without them. Some are even lost with them. But how do you get up-to-date data books?

Some distributors will sell data books. They are reasonably priced in the US since shipping is usually the only expense to the distributor. Canada customs however can not understand that companies give these thick books away. They decide how much they might be worth and charge tax and duty accordingly. Buying them from the Canadian distributor is the easy way, but you will them often over priced and out of date. The newer ones being saved for the "big" customers.

Do you want one for free? Phone and ask for one. Real tough, eh? Get the number for the company's main office and ask for the literature department, marketing, or admit you want a

Paul Chidley is a senior technologist at NovAtel, an Alberta based cellular phone company. He's a neophyte ZCPR user, but has been active in homebrewed hardware and software design for many years, primarily in the Ohio Scientific and 6502/816 area. Paul can be reached on GEnie (email address: P.CHIDLEY), by regular mail at 162 Hunterhorn Drive NE, Calgary Alberta, Canada, T2K 6H5, or by phone at (403)274-8891 during reasonable MST hours.

data book and see where you get transferred.

Rule Number One: Never tell anyone you are a student or hobbyist, especially if you are. Big companies are often very short sighted. To them a student is a person in school, someone that will never actually have a job and place a big order with them. Even if you are going for your PhD in electronics you are still a "student." A "hobbyist" is even worse, this is someone that builds Heathkits, reads *Popular Electronics* and has a pocket protector for every day of the week! Big companies don't realize that many "hobbyists" do this for a living. Even if your day job involves superconductors at MIT, at night you are only a "hobbyist."

So what are you? Well, if you tinker in hardware then you are a "hardware systems designer." If software is your game then how about a "software consultant" or "embedded controller programmer" or even a "C mercenary!" (Don't mention Forth. They won't know what it is!) Now I'm not saying you should lie. If you do, their next question could need a bigger one for an answer. Tell the truth but tell them what they want to hear. When they ask for the company name you can give your day job address as long as it is something suitable like "WKTV Idaho," or "Ingenious Devices Incorporated," and not "Cockrane Cattle Feed and Farm Supplies." Or use your home address and say you are a consultant, free-lancer, or that you are currently between assignments, i.e. an unemployed engineer. What ever is most suitable.

There are also manufacturer representatives. I'm not exactly sure, in this age of telecommunications, where a manufacturer rep fits in. I have asked several and they are not sure either. If you have no luck with the manufacturer you can always try the rep. These are listed in the data book—the one you don't have. So you're back to calling head office again anyway.

What data books do you want? Check the ads in trade magazines. Trade magazines? I think that better be next issue's topic. I recently answered an ad from IDT to get a data sheet on their new fact logic parts. I left my name and such on their answering machine along with what I was looking for. A week later it arrived in the mail. The next day a box arrived, a big box, with their complete set of 1992 data books. Trade magazines are a gold mine for finding what is available on the market.

Now that you have some idea how to get a data book make sure you get one from Dallas Semiconductor. If you are a true technomaniac you will get many hours of enjoyment discovering some of their truly unique products. You will be pleased to learn that they provide a parts hot line. You can order any Dallas Semiconductor product for quick shipment by using a Visa, MasterCard or American Express. Call and ask for their Overnight Delivery Service at 800-336-6933. If you are calling from outside North America, dial 214-450-5351. Or you can fax your order to them at 214-450-0470. Every electronics company should sell parts this way. Many companies have a lot to learn about marketing from these fellows.

Next time

Anyway enough is enough. If you liked this article let our editor know and I'll write some more. If you hated it let him know and I'm off the hook. If you caught any spelling or grammatical errors, tough. I is a techy, failed English. I'm only a technologist but I have heard that good engineers are lousy spellers so I must be doing OK! See you all in Trenton (I hope).●

Addresses of firms mentioned:

Dallas Semiconductor
4401 South Beltwood Parkway
Dallas, Texas 75244-3292
(214) 450-0400

Editor, from page 2

suspend tasks and do several functions similar to SideKick, though it predated the MS-DOS utility. From there, he wrote BackGrounder ii, a full task swapper. Jay Sage describes BGii as "the most spectacular piece of CP/M programming that has ever been written. Just incredible."

DOSDisk is another of Bridger's products. It maps MSDOS file structure into CP/M and is the only disk emulator to handle directory trees.

Trenton 1992

For the last seventeen years, spring has marked the gathering of the micro computing faithful to a college campus mid-way between Princeton and Philadelphia. Some 15,000 attend. Most come from a region spanning Boston to Washington DC, and west to Ohio and Tennessee, though occasional "Left Coasters" wander in.

This event, known as the Trenton Computer Festival, has become the annual event for CP/M afficiados. This breed does not limit its reach to the northeastern United States. Alberta, Washington State, California—the entire continent of North America seems to send 8-bitters to Trenton!

Spring returns. Again you can see birds return to their nests and CP/Mers return to their roots. Almost sounds like the practice of a strange cult, eh? Make no mistake about it, Trenton, the event, is nearly here.

Ah, what an event! A legendary fleamarket where anything that was ever put in a computer case can be bought for a song. One *TCJ* reader needed a flatbed truck to carry the VAX 730 just purchased to fit in the spare bedroom. (True story, except it didn't fit. The fellow had to haul it to the basement with a crew of five guys and a small crane). Imsai's are frequently sighted.

Ah, what an event! Hundreds upon hundreds of commercial booths. If there is a company that sells it, you'll find it at Trenton.

Ah, what an event! There are seven programs of special interest groups running concurrently! You would have to bring Uncle Harry and the kids to cover all the meetings! The CP/Mers hold their yearly Gathering of the Faithful all day Saturday and not a minute is wasted.

See Editor, page 36

What Zilog Never Told You About The Super8

Brad and Doug's Excellent Adventure

By Brad Rodriguez, T-Recursive Technology, Toronto, Ontario
and Doug Fleenor, Doug Fleenor Design, Arroyo Grande, California

Reset

The Super8 is very finicky about its reset signal. A Zilog memo says "If the Super8 reset pulse is shorter than specification, or if V_{CC} is not stable throughout the reset sequence, the Super8 may latch up, requiring power down."

Zilog says this has been fixed in the rev A part, but we've had problems with it, too. Although rev A is much improved.

In general, we've had bad luck with RC reset circuits. We use a MAX690 or equivalent "microprocessor supervisory circuit" to generate our RESET\ signal.

Initialization

The Super8 is rather particular about the sequence of initializing its control registers, and some of the Zilog-supplied demo files are actually incorrect. Particular notes:

- It seems necessary to initialize the PM register before the P0M register.
- It also seems necessary to allow a short time delay — 3 NOPs — after changing PM or P0M.
- When using P05-P07 as address bits (ROMless parts only): since Port 0 is enabled as all outputs BEFORE it is enabled as an address bus, you *must* write the high byte of the program counter to Port 0 *before* setting PM or P0M. (Usually this high byte is 00 hex, depending on where your initialization code is located.)

Our recommended initialization sequence:

```
ld    p0,#^HB($)  ; output high adrs byte to P0
ld    pm,#20h     ; P1 adr/dta, P0 output, push-pull
nop
nop
nop
ld    p0m,#0ffh   ; enable 16 bit address output
nop
nop
nop
```

Clock

We've had bad luck deriving a clean 20 MHz external clock from the on-chip oscillator. When we need the 20 MHz clock elsewhere in the system, we use a TTL oscillator tied to XTAL1.

High Address Bits

A word of warning about EPROM emulators: many Super8 designs use weak (~100K ohm) pulldowns on A13-A15, to make these address bits zero until the initialization code takes effect. Many EPROM emulators have pullup resistors on these lines! It may be necessary to remove the pullups from your EPROM emulator in order for it to work in your Super8 circuit.

WAIT\ (on P34)

Zilog issued a memo about WAIT\ timing. It says:

- WAIT\ must be asserted low within 90 nsec of AS\ going high.
- WAIT\ must have a rise time of 20 nsec maximum.
- The fall time of WAIT\ is not critical, as long as it meets the 90 nsec requirement.
- WAIT\ going high must be stable within 20 nsec of XTAL1 going high.

To be safe, WAIT\ should be synchronized with a flip-flop clocked by the falling edge of XTAL1. This will also give the required rise time.

DM\ (on P35)

You must enable P35 as an output bit (in P2CM), for DM\ to work.

Counter/Timers

You should stop a counter/timer before reloading it. Failure to do so can cause invalid data to be loaded occasionally. Zilog says this is fixed on the rev A parts; we haven't tested.

Zilog doesn't say much about the counter/timer output pin. We've found by experimentation (rev 0 parts):

- When the counter is disabled, the counter output pin (if used) is set to zero.
- When running continuously, the output toggles at end of count.
- When the Bi-value mode is initialized, the output is zero. So, the Time Constant register holds the length of the low output, and the Capture register holds the length of the high output.
- When running in single cycle mode, we observe a high pulse of about 100 nsec duration at end of count. Our supposition is that the end of count toggles the bit high, immediately followed by a "counter disable" which forces the bit back low.

UART

The wake-up function doesn't work as advertised on rev 0 parts. (rev A is OK.) We will have more details forthcoming.

Using CLR UIO to send a null character, also clears the receive character available flag. Use LD UIO,#0 instead.

Interrupts

DI (global interrupt disable) does *not* disable fast interrupts! The Super8 Tech Manual says: "when fast interrupt processing is used, the IMR bit for the selected level must also be set." In fact, fast interrupts seem to ignore IMR.

DMA

The DMA address register is physical register pair %C0-%C1, not working register RR0. This is unclear in the Zilog Tech Manual. Relocating the working registers away from %C0 allows RR0 to be used without disturbing the DMA address.

When doing DMA INPUT through port 4, handshake channel 0, strobed mode, we have found this setup sequence necessary:

- set up DMA address to the desired address, and DMA count to the desired count - 1.
- enable handshake channel 0.
- read and discard port 4.
- *Then* enable DMA.

The "read and discard" operation seems to be necessary only when switching port 4 from output to input. Our supposition is that data strobed out on port 4 is also latched as a valid input byte, which must then be discarded.

When doing DMA OUTPUT through port 4, handshake channel 0, strobed mode, we have found this setup sequence necessary:

- set up DMA address to the desired address + 1, and DMA count to the desired count - 2.
- enable handshake channel 0.
- write the first byte of data manually to port 4.
- *Then* enable DMA.

In particular, enabling the DMA before writing to port 4 seems to give incorrect results. (The Zilog manual says you have to write the first byte manually, but implies that you should enable DMA first.)

Assembler and Linker

When assembling indexed-addressing LDE and LDC instructions, when the offset is between 128 and 255 decimal, the assembler *incorrectly* uses the short offset form. (The Super8 uses one-byte *signed* short offsets.) You can force the correct long-offset form by applying the ^WORD operator to the offset. Example:

`LDE R6,192(RR8)` should be written `LDE R6,^WORD(192)(RR8)`

Note: ^WORD also works with symbolic labels.

The assembler requires the forms "3FH" or "3FX" or "%3F" for hex numbers (using 3F as an example). However, "3F" is *not* reported as an error, and does *not* assemble the correct value. Be wary of hex numbers without a base indicator!

Do *not* use relocatable symbolic labels with the SRP instructions (e.g., SRP0 #WKGREGS where WKGREGS is defined in a relocatable .SECTION). The linker cannot relocate these, and you get SRPx #0.

The linker's rules for combining like-named .SECTIONs are not clear, and in some cases we believe the manual is incorrect. We use named sections to group related code, e.g., we will declare a PROM section and a RAM section, and switch back and forth in the assembly source file. We have found that the first appearance of the name should specify the attributes, including the "C" (concatenate) option. All subsequent appearances should have only the "C" attribute. Example:

```
.SECTION PROM,A,X,C
   ...
   ...
   ...
```

Omitting the "C" attribute will cause the linker to attempt to put all of the like-named sections at the same address!

Debugger

Be advised that the debugger uses RAM from F000 to FFFF, and many registers. It also traps the UART receive interrupt.

We have found that the debugger really can't access physical registers C0 to CF. When debugging, you should locate your working registers somewhere else. We usually use SRP #0.

Despite what the manual says, the debugger does not access External memory when the X option is used. This is a bug in the code. The debugger also does not enable the DM\ output, even when the X option is used.

Bank 0 must be active ("SB0") when a breakpoint is encountered. Bank 1 will cause a crash.

The breakpoint resets the stack pointer to a private stack, so you can't determine where your stack pointer was when the breakpoint was encountered. (The stack pointer is correctly restored when execution is continued, so it must be saved somewhere.)●

Editor, from page 34

Ah, what an event! A banquet on Saturday night. This year the 8-bitters are planning their own, complete with evening working sessions at the hotel.

Ah, what an event! If you're an 8-bitter, you need to be there. Mark it on your calendar: April 11 and 12 at the Mercer County Community College, Mercer NJ. Call Lee Bradley at (203) 666-3139 or log on his Z-Node at (203) 665-1100.

Ah, what an event!

Administrata

A little bit of business news for those who may have an interest. *TCJ* completed its ninth year of publication, and its first under the current publisher, with circulation up 27%. We published 328 pages in 1991, against 236 in 1990. By moving advertising to the covers, we increased editorial content over 50%. While success at *TCJ* is not measured in the traditional business sense but in the involvement of its readers, we completed the year in a stronger position. This bodes well for the future of grassroots computer publishing, particularly as it was accomplished during a deep recession.

Measuring our success by the involvement of the readers, we had an even better year! Our *Reader-to-Reader* column has

See Editor, page 43

An Arbitrary Waveform Generator

Using the Harris RTX2001A

By Jan Hofland

[We conclude this three part series on the building of a waveform generator for structural testing with the full source code. Please refer to issues 52 and 53 for hardware, schematics and Forth extensions.—Ed.]

Operating Source Code

```
HEX
4300 DUP H ! H-FENCE !
67FE H-TOP !
6800 DUP R ! R-FENCE !
8FFE R-TOP !

The following definition is based on the definition of U< and
is here because it's faster than the signed version of >

: U>  ( u1 u2 – flg )        returns TRUE if unsigned u1 is
       SWAP- DROP             greater than unsigned u2, else
       0 0 -c ;               return FALSE

: UMAX  ( u1 u2 – u )        unsigned version of MAX
       2DUP U<
       IF NIP                 if u1 < u2 keep u2
       ELSE DROP              otherwise keep u1
       THEN ;

: UMIN  ( u1 u2 – u )         unsigned version of MIN
       2DUP U< IF DROP ELSE NIP THEN ;

Some low level operations for the external devices added to
the ASIC bus

: DAC!  ( n – )              write 16 bit value to the DAC
       18 G! ; MACRO          shift register

: fifo@  ( – n )             fetch a value from the input
       19 G@ ; MACRO          fifo

: filtCnt!  ( n – )          store an 8 bit value to the
                              filter clock load register.
                              Actual value can be 16 bits
       1E G! ;                Only the lower 8 bits are used.

: cntrEnabl  ( – )           enable the filter clock counter.
                              This is a form of a pseudo-
                              register. The data written
       0 1C G! ;              doesn't matter.

: cntrDisabl  ( – )          disable the filter clock counter
       0 1D G! ;

: toglFC  ( – )              toggle the filter clock output.
                              A pseudo register.Used when the
                              filter clock is under software
       0 1F G! ;              control.

: fifoMT?  ( – flg )         return zero if fifo is not
       1B G@ 1 AND ; MACRO    empty, non-zeroif it is empty

: fifoFul?  ( – flg )        return zero if the fifo is not
       1B G@ 2 AND ; MACRO    full, non-zero if it is full

This code sets the interrupt routine for the non-maskable
interrupt. A pushbutton switch is connected to NMI to abort
the current process

: pbStop  ( – )              NMI Interrupt service routine
       TRUE ABORT" Stopped" ;

Install the routine

The following GOES> and DOES> definitions courtesy Jack Woehr

: _DOES  R> U2/ USE ;

: GOES>  COMPILE _DOES BE01 , COMPILE @ 1 -OPT ! ; IMMEDIATE

: DOES>  COMPILE _DOES BE01 ,          1 -OPT ! ; IMMEDIATE

: table  ( – )  CREATE DOES> SWAP CELLS + @ ;  define a table
                                               ;in code space
        ( n1 – n2 runtime ) returns value of nth entry at
                            runtime table sine build a 800
                            point sine wave
                            table. amplitude 28284 peak
0000 , 00DE , 01BC , 029A , 0378 , 0456 , 0534 , 0612 ,
06EF , 07CD , 08AB , 0988 , 0A65 , 0B42 , 0C1F , 0CFC ,
0DD8 , 0EB5 , 0F91 , 106D , 1148 , 1223 , 12FE , 13D9 ,
14B3 , 158D , 1667 , 1740 , 1819 , 18F2 , 19CA , 1AA2 ,
1B79 , 1C50 , 1D27 , 1DFD , 1ED2 , 1FA8 , 207C , 2150 ,
2224 , 22F7 , 23C9 , 249B , 256C , 263D , 270D , 27DD ,
28AC , 297A , 2A47 , 2B14 , 2BE0 , 2CAC , 2D77 , 2E41 ,
2F0A , 2FD3 , 309B , 3162 , 3228 , 32EE , 33B2 , 3476 ,
3539 , 35FC , 36BD , 377E , 383D , 38FC , 39BA , 3A77 ,
3B33 , 3BEE , 3CA8 , 3D61 , 3E19 , 3ED1 , 3F87 , 403C ,
40F0 , 41A4 , 4256 , 4307 , 43B7 , 4466 , 4514 , 45C1 ,
466C , 4717 , 47C0 , 4869 , 4910 , 49B6 , 4A5B , 4AFF ,
4BA1 , 4C43 , 4CE3 , 4D82 , 4E1F , 4EBC , 4F57 , 4FF1 ,
508A , 5121 , 51B7 , 524C , 52E0 , 5372 , 5403 , 5492 ,
5521 , 55AE , 5639 , 56C3 , 574C , 57D4 , 585A , 58DE ,
5962 , 59E4 , 5A64 , 5AE3 , 5B61 , 5BDD , 5C57 , 5CD1 ,
5D48 , 5DBF , 5E34 , 5EA7 , 5F19 , 5F89 , 5FF8 , 6065 ,
60D1 , 613B , 61A4 , 620B , 6271 , 62D5 , 6337 , 6398 ,
63F8 , 6455 , 64B2 , 650C , 6565 , 65BD , 6612 , 6667 ,
66B9 , 670A , 675A , 67A7 , 67F3 , 683E , 6887 , 68CE ,
6913 , 6957 , 6999 , 69DA , 6A18 , 6A56 , 6A91 , 6ACB ,
```

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

As the Internal Revenue Service sees it, the United States is a land of untold wealth....

```
6B03 , 6B39 , 6B6E , 6BA1 , 6BD2 , 6C02 , 6C30 , 6C5C ,
6C87 , 6CAF , 6CD6 , 6CFC , 6D1F , 6D41 , 6D61 , 6D80 ,
6D9C , 6DB7 , 6DD1 , 6DE8 , 6DFE , 6E12 , 6E24 , 6E35 ,
6E44 , 6E51 , 6E5C , 6E66 , 6E6E , 6E74 , 6E78 , 6E7B ,

6E7C , 6E7B , 6E78 , 6E74 , 6E6E , 6E66 , 6E5C , 6E51 ,
6E44 , 6E35 , 6E24 , 6E12 , 6DFE , 6DE8 , 6DD1 , 6DB7 ,
6D9C , 6D80 , 6D61 , 6D41 , 6D1F , 6CFC , 6CD6 , 6CAF ,
6C87 , 6C5C , 6C30 , 6C02 , 6BD2 , 6BA1 , 6B6E , 6B39 ,
6B03 , 6ACB , 6A91 , 6A56 , 6A18 , 69DA , 6999 , 6957 ,
6913 , 68CE , 6887 , 683E , 67F3 , 67A7 , 675A , 670A ,
66B9 , 6667 , 6613 , 65BD , 6565 , 650C , 64B2 , 6455 ,
63F8 , 6398 , 6337 , 62D5 , 6271 , 620B , 61A4 , 613B ,
60D1 , 6065 , 5FF8 , 5F89 , 5F19 , 5EA7 , 5E34 , 5DBF ,
5D48 , 5CD1 , 5C58 , 5BDD , 5B61 , 5AE3 , 5A64 , 59E4 ,
5962 , 58DE , 585A , 57D4 , 574C , 56C3 , 5639 , 55AE ,
5521 , 5492 , 5403 , 5372 , 52E0 , 524C , 51B7 , 5121 ,
508A , 4FF1 , 4F57 , 4EBC , 4E1F , 4D82 , 4CE3 , 4C43 ,
4BA1 , 4AFF , 4A5B , 49B6 , 4910 , 4869 , 47C1 , 4717 ,
466C , 45C1 , 4514 , 4466 , 43B7 , 4307 , 4256 , 41A4 ,
40F0 , 403C , 3F87 , 3ED1 , 3E1A , 3D61 , 3CA8 , 3BEE ,
3B33 , 3A77 , 39BA , 38FC , 383D , 377E , 36BD , 35FC ,
3539 , 3476 , 33B2 , 32EE , 3228 , 3162 , 309B , 2FD3 ,
2F0A , 2E41 , 2D77 , 2CAC , 2BE0 , 2B14 , 2A47 , 297A ,
28AC , 27DD , 270D , 263D , 256C , 249B , 23C9 , 22F7 ,
2224 , 2150 , 207C , 1FA8 , 1ED3 , 1DFD , 1D27 , 1C50 ,
1B7A , 1AA2 , 19CA , 18F2 , 181A , 1741 , 1667 , 158E ,
14B3 , 13D9 , 12FE , 1223 , 114B , 106D , 0F91 , 0EB5 ,
0DD8 , 0CFC , 0C1F , 0B42 , 0A65 , 0988 , 08AB , 07CD ,
06F0 , 0612 , 0534 , 0456 , 0378 , 029A , 01BC , 00DE ,

0000 , FF22 , FE44 , FD66 , FC88 , FBAA , FACC , F9EE ,
F911 , F833 , F755 , F678 , F59B , F4BE , F3E1 , F304 ,
F228 , F14B , F06F , EF94 , EEB8 , EDDD , ED02 , EC27 ,
EB4D , EA73 , E999 , E8C0 , E7E7 , E70E , E636 , E55E ,
E487 , E3B0 , E2D9 , E203 , E12E , E059 , DF84 , DEB0 ,
DDDC , DD09 , DC37 , DB65 , DA94 , D9C3 , D8F3 , D823 ,
D755 , D686 , D5B9 , D4EC , D420 , D354 , D289 , D1BF ,
D0F6 , D02D , CF65 , CE9E , CDD8 , CD12 , CC4E , CB8A ,
CAC7 , CA04 , C943 , C882 , C7C3 , C704 , C646 , C589 ,
C4CD , C412 , C358 , C29F , C1E7 , C12F , C079 , BFC4 ,
BF10 , BE5C , BDAA , BCF9 , BC49 , BB9A , BAEC , BA3F ,
B994 , B8E9 , B840 , B797 , B6F0 , B64A , B5A5 , B501 ,
B45F , B3BE , B31D , B27E , B1E1 , B144 , B0A9 , B00F ,
AF76 , AEDF , AE49 , ADB4 , AD20 , AC8E , ABFD , AB6E ,
AADF , AA52 , A9C7 , A93D , A8B4 , A82C , A7A6 , A722 ,
A69E , A61C , A59C , A51D , A49F , A423 , A3A9 , A32F ,
A2B8 , A241 , A1CC , A159 , A0E7 , A077 , A008 , 9F9B ,
9F2F , 9EC5 , 9E5C , 9DF5 , 9D8F , 9D2B , 9CC9 , 9C68 ,
9C08 , 9BAB , 9B4E , 9AF4 , 9A9B , 9A43 , 99EE , 9999 ,
9947 , 98F6 , 98A6 , 9859 , 980D , 97C2 , 9779 , 9732 ,
96ED , 96A9 , 9667 , 9626 , 95E8 , 95AA , 956F , 9535 ,
94FD , 94C7 , 9492 , 945F , 942E , 93FE , 93D0 , 93A4 ,
937A , 9351 , 932A , 9304 , 92E1 , 92BF , 929F , 9280 ,
9264 , 9249 , 922F , 9218 , 9202 , 91EE , 91DC , 91CB ,
91BC , 91AF , 91A4 , 919A , 9192 , 918C , 9188 , 9185 ,

9184 , 9185 , 9188 , 918C , 9192 , 919A , 91A4 , 91AF ,
91BC , 91CB , 91DC , 91EE , 9202 , 9218 , 922F , 9249 ,
9264 , 9280 , 929F , 92BF , 92E1 , 9304 , 932A , 9351 ,
9379 , 93A4 , 93D0 , 93FE , 942E , 945F , 9492 , 94C7 ,
94FD , 9535 , 956F , 95AA , 95E8 , 9626 , 9667 , 96A9 ,
96ED , 9732 , 9779 , 97C2 , 980D , 9859 , 98A6 , 98F6 ,
9947 , 9999 , 99ED , 9A43 , 9A9B , 9AF4 , 9B4E , 9BAB ,
9C08 , 9C68 , 9CC9 , 9D2B , 9D8F , 9DF5 , 9E5C , 9EC5 ,
9F2F , 9F9B , A008 , A077 , A0E7 , A159 , A1CC , A241 ,
A2B7 , A32F , A3A8 , A423 , A49F , A51D , A59C , A61C ,
A69E , A721 , A7A6 , A82C , A8B4 , A93D , A9C7 , AA52 ,
AADF , AB6D , ABFD , AC8E , AD20 , ADB4 , AE49 , AEDF ,
AF76 , B00F , B0A9 , B144 , B1E1 , B27E , B31D , B3BD ,
B45F , B501 , B5A5 , B64A , B6F0 , B797 , B83F , B8E9 ,
B993 , BA3F , BAEC , BB9A , BC49 , BCF9 , BDAA , BE5C ,
BF0F , BFC4 , C079 , C12F , C1E6 , C29F , C358 , C412 ,
C4CD , C589 , C646 , C704 , C7C3 , C882 , C943 , CA04 ,
CAC6 , CB8A , CC4D , CD12 , CDD8 , CE9E , CF65 , D02D ,
D0F6 , D1BF , D289 , D354 , D41F , D4EC , D5B9 , D686 ,
D754 , D823 , D8F3 , D9C3 , DA94 , DB65 , DC37 , DD09 ,
DDDC , DEB0 , DF84 , E058 , E12D , E203 , E2D9 , E3AF ,
E486 , E55E , E636 , E70E , E7E6 , EBBF , E999 , EA72 ,
EB4C , EC27 , ED02 , EDDC , EEB8 , EF93 , F06F , F14B ,
F227 , F304 , F3E1 , F4BE , F59B , F678 , F755 , F833 ,
F910 , F9EE , FACC , FBAA , FC88 , FD66 , FE44 , FF22 ,

VARIABLE phase                  sinewave phase range 0..799
VARIABLE phaseInc               phase increment between points
VARIABLE delPhase               intermediate value between phase
                                points used only for high
                                accuracy sine mode
VARIABLE delPhaseInc            delPhase increment between points

DECIMAL
800 CONSTANT twoPi              table index corresponding to 2
                                pi radians
200 CONSTANT pi/2

Using scratchpad register RH to hold the next data to load
into the DAC and scratchpad RX to hold a flag that denotes
whether the data in RH is new or has already been sent to the
DAC.The flag will be cleared when data is loaded into RH and
set when it'e transferred to the DAC

: >RH  ( n - )                  transfer n to RH & clear RX flag
        RH! FALSE RX! ; MACRO

: phaseAdj  ( n1 - n2 )         adjust phase if it exceeds 1
        DUP twoPi                cycle
        U< IF EXIT THEN twoPi - ;  this was the fasteat
                                 if..then

: nextPt  ( - )                 put next sinewave point in
                                 scratchpad and then update phase
        phase @ DUP sine >RH    fetch sine from table & store it
        phaseInc @ +            fetch phase increment and add
        phaseAdj                adjust to keep within one cycle
        phase ! ;               update phase

: newPt  ( - )                  output data to DAC from
        RH@ DAC! RX- ;          scratchpad RH and set RX flag

Accurate Sine Mode

Will interpolate between sine table points according to the
formula sin(x+delta_x) = sin(x) * cos(delta_x) + sin(delta_x)
* cos(x) for small angles cos(delta_x) about equals 1 and
sin(delta_x) is approximately delta_x, where x is in radians.
Thus, sin(x+delta_x) will be approximated by sin(x) +
delta_x*cos(x). Will scale the result in 50 mVrms increments
and subtract a constant offset.

VARIABLE offset                 subtracted from scaled sine
                                value
VARIABLE scale                  a value between 1 & scaleTop
40 CONSTANT scaleTop            the sinetable points correspond
                                to an output amplitude of 2
                                Vrms. 1/40th of this value
                                corresponds to 50 mVrms

: newPhase  ( - )               calculate next phase and
                                 delPhase from phaseInc and
                                 delPhaseInc
        delPhase @ delPhaseInc @ +   the part between table
        twoPi /MOD              points does it bump up into next
        SWAP delPhase !         point the new delPhase value
        phase @ + phaseInc @ + add carryover plus phaseInc to
                                phase
        phaseAdj                adjust if it goes over one cycle
        phase ! ;

: div512  ( n1 - n2 )           divide top of stack by 512
        DUP 0< IF               want to avoid division of small
                                numbers
        NEGATE 2/ 2/ 2/ 2/ 2/ 2/ 2/ 2/ 2/ NEGATE returning -1
        ELSE   2/ 2/ 2/ 2/ 2/ 2/ 2/ 2/ 2/
        THEN ;

: cosAdj  ( - n )               return the cosine correction for
                                 the current values of phase and
                                 delPhase
        phase @ pi/2 +          sin(x+pi/2) = cos(x)
        phaseAdj                keep it within one cycle
        sine                    cosine
        delPhase @ 995 */       512 times delta_x
```

```
        div512 ;                divide by 512

: accuPt  ( — n )               return sine of current phase
                                 scaled by scale/scaleTop and
                                 subtract offset
        phase @ sine            current sine
        cosAdj +                cosine correction
        scale @ scaleTop */     apply scale factor
        offset @ - ;            and subtract offset

: newAccuPt  ( — )              load sine point into the DAC
        accuPt >RH              scratchpad and then
        newPhase ;              update phase
GUARD

HEX
: initTimers  ( — )             set all three timers for
        IBC@ FCFF AND IBC!      internal clock
        TIMER0 MASK             and mask off all three
        TIMER1 MASK             interrupts
        TIMER2 MASK ;
```

Set up timer 0 and timer 1 interrupt vectors

```
: intSetup  ( — )               move copy of service routines
                                 into interrupt vector space
        ['] newPt               DAC load interrupt
        7 !INTERRUPT            level 7 interrupt
        ['] toglFC              filter clock toggle interrupt
        8 !INTERRUPT ;          level 8 interrupt
```

Set Filter Cutoff Point

The lowpass filter cutoff is controlled by a clock that is 100 times the selected lowpass frequency. For wide bandwidths, it is controlled by a programmable hardware down counter connected to the ASIC bus. Whenever the counter reaches terminal count, the filter clock is toggled. For lower bandwidths, the toggle rate is controlled by interrupts from timer 1. The crossover is for a period of 256 counts, corresponding to a cutoff frequency of about 160 Hz. For wide bandwidths, the cutoff is quantized to some rather wide values, in that the toggle period must be an integer.

```
DECIMAL
40000 CONSTANT maxBW            upper bandwidth limit
    1 CONSTANT minBW            lower bandwidth limit
VARIABLE cutoff

: setCutoff  ( n — )            set filter cutoff to about u Hz
        maxBW UMIN minBW UMAX   check limits
        DUP cutoff !            store a copy of the limited
                                 value
        maxBW 0 ROT UM/MOD NIP 1- calculate toggle period
        DUP 256                 if the interval is less than
        U< IF                   256, use the hardware counter
          filtCnt!              store period in counter load
                                 register
          TIMER1 MASK           mask off timer 1 interrupts
          cntrEnabl             and enable the hardware counter
        ELSE                    otherwise, use timer 2 for
                                 interval
          TC1!                  store period in timer 1
          cntrDisabl            disable the hardware counter
          TIMER1 UNMASK         and unmask the timer 1 interrupt
        THEN ;
```

Set Timebase and Phase Increment for Sinewave Generation

The relationship between phase increment, timer 0 interval (the timebase) and the desired frequency is:

phase increment = ((#table_points per cycle) * (timer 0 period) * frequency) / timer 0 clock frequency

Set up a table of minimum timebase intervals for each of the five operating modes

```
table minIval
         79 ,   mode 0 fast sine  100 KHz sample rate
        431 ,   mode 1 accurate sine  18.5 KHz sample rate
         79 ,   mode 2 circular buffer 100 KHz sample rate
        132 ,   mode 3 ping pong queue  60 KHz sample rate
         79 ,   mode 4 once through queue 100 KHz sample rate
```

The minimum timebase intervals have been extended to accomodate software simulation of the hardware command queue

```
         95 ,   mode 0 fast sine 83.3 KHz sample rate
        455 ,   mode 1 accurate sine  17.5 KHz sample rate
         95 ,   mode 2 circular buffer 83.3 KHz sample rate
        199 ,   mode 3 ping pong queue 50 KHz sample rate
         99 ,   mode 4 once through queue 80 KHz sample rate

VARIABLE mode                   what is the operating mode
VARIABLE sineFreq               output frequency
VARIABLE tbPeriod               actual timer 0 preload value
table maxFreq 40000 , 7200 ,    upper limit for output frequency
                                 mode dependent
mode @ minIval tbPeriod !       initialize interval

: calcInt  ( phaseInc delPhaseInc — )  calculate timebase
                                 interval from phase increment
                                 and sinewave frequency
        25 2 */ S>D             fractional phase increment
                                 contribution
        ROT 10000 UM* D+        integer phase increment
                                 contribution
        sineFreq @ UM/MOD       divide by sine frequency
        SWAP 2* sineFreq @ / + 1-  round result and subtract 1
        mode @ minIval UMAX     make sure it's at least minimum
        tbPeriod ! ;            store result

: calcPhaseInc  ( freq — )      calculate phase increment from
                                 frequency
        tbPeriod @ 1+ UM*       multiply by timebase period
        10000 UM/MOD            divide by timebase clock ratio
                                 to # of phase points per cycle
        SWAP 2 25 */            fractional phase adjust range to
                                 {0..799}
        mode @ 0=               mode 0 ?
        IF                      yes
          400 / +               increment phase
                                 Inc if fractional part > 1/2
          1 MAX                 must be at least 1
          0                     mode 0 delPhaseInc
        THEN
        2DUP delPhaseInc !      store fractional phase increment
        phaseInc !              and the integer part
        calcInt ;               must now recalculate the
                                 timebase interval to account for
                                 having to set the phase
                                 increment to a non-zero value
```

Calculate the actual frequency given the timebase interval and the phase increment. Presumes that a valid value has been stored in interval.

```
: calcFreq  ( — )               calculate actual frequency from
                                 phase increment & timebase
                                 interval
        phaseInc @ 10000 UM*
        delPhaseInc @ 25 2 */ S>D D+
        tbPeriod @ 1+ UM/MOD
        sineFreq ! DROP ;

: setFreq  ( freq — )           set up sine freq, sample rate
        mode @ minIval tbPeriod ! first set minimum sample rate
        mode @ maxFreq UMIN     max frequency is mode dependent
        DUP sineFreq !          store the value
        calcPhaseInc            and calculate phase increment
        tbPeriod @ TC0!         store interval in timer 0 load
                                 register
        calcFreq ;              calculate actual frequency from
                                 interval and phase increment

: setPeriod  ( period — )       set sample rate
        mode @ minIval UMAX     check against mode dependent min
        mode @                  if mode 0 or mode 1 (sine
        0 2 WITHIN              output), then need to adjust
                                 phase increment
        IF                      find maximum interval based on
          3200000.              2-1/2 points per cycle
```

```
        sineFreq @ UM/MOD NIP 1-
        UMIN tbPeriod !     keep the smaller period
        sineFreq @ calcPhaseInc  and calculate new phase
                            increments
        calcFreq
      ELSE tbPeriod !       not sine mode so just store
                            period
      THEN
      tbPeriod @ TC0! ;     finally, set the timer 0 load
                            register
GUARD
```

Data Buffer Setup

Going to use two circular data queues for arbitrary data input and output. The queues can be of any length up to a maximum of 2K words. They can be used in a circular fashion, repeating the data sequence, or they can be used in a first-in first-out manner with non-repeating data. They can also be used in a ping pong fashion where data is flushed first from one and then the other.

Set up to build queue structures in code space and in data memory space. The underlying routines can be common to both. The only differences have to do with allocation of memory. The structure consists of a cell containing the number of values, a tail pointer where new items are inserted, a head pointer where items are removed from the queue, and an end pointer keeping track of the last cell in the storage area.

```
DECIMAL

: clearQ  ( addr – )          given the start address of a
                               queue structure clear the
                               #values variable and initialize
                               the head and tail pointers.
      0 OVER !                 initialize #values
      DUP 8 + DUP ROT 2+
      !+2 ! ;                  initialize the tailPtr &
                               headPtr

: buildCQ  ( n <name> – )     build a queue in code memory
                               space of n cells
      CREATE                   create the dictionary head
      HERE >R                  save start address
      4 + CELLS DUP            total number of bytes for
                               structure
      GAP                      allocate data storage space
      R@ clearQ                initialize #values, headPtr,
                               tailPtr
      R@ + 2- R> 6 + ! ;       set endPtr

: buildDQ  ( n <name> – )     build an n cell queue in data
                               memory space
      THERE >R                 save start address
      VARIABLE                 create the dicitonary head and
                               allocate the first cell
      3 + CELLS DUP            total number of bytes minus 2
                               for structure
      ALLOT                    allocate data storage space
      R@ clearQ                initialize #values, headPtr,
                               tailPtr
      R@ + R> 6 + ! ;          set endPtr

: >Cbuf  ( n addr – )         insert a value into the queue
                               overwriting the existing value
      DUP >R                   save buffer address
      2+ @                     get tailPtr
      !+2                      store value & increment pointer
      DUP                      check pointer upper bound
      R@ 6 + @                 get endPtr
      U> IF                    adjust if tailPtr past endPtr
        DROP                   discard out of bounds pointer
        R@ 8 +                 and make it point to the first cell
        THEN
      R@ 2+ !                  store tailPtr
      1 R> +! ;                increment #values

: Cbuf>  ( addr – n )         extract next value from the
                               queue
      DUP >R                   save buffer address
      4 + @                    get headPtr
      @+2                      get value & increment pointer
      DUP                      check pointer upper bound
      R@ 6 + @                 get endPtr
      U> IF                    adjust if tailPtr past endPtr
        DROP                   discard out of bounds pointer
        R@ 8 +                 and make it point to the first
                               cell
        THEN
      R@ 4 + !                 store headPtr
      -1 R> +! ;               decrement #values

: >Que  ( n addr – fflg )     put n into the queue & return a
                               FALSE flag if it isn't full.
                               Otherwise return the value
      ( n addr – n addr tflg ) & address & TRUE flag.
      DUP @ CELLS              get #values
      OVER 6 + DUP @ SWAP-     calculate queue size from endPtr
      U< IF >Cbuf FALSE        there is room if 2x #values
        ELSE TRUE              less than size
        THEN ;

: Que>  ( addr – n tflg )     fetch a value from the queue &
                               return a TRUE flag if the queue
                               isn't empty. Otherwise,
      ( addr – fflg )          return FALSE flag.
      DUP @                    get #values
      IF Cbuf> TRUE
      ELSE DROP FALSE
      THEN ;

: peeQue  ( <name> – n )      return value at head of queue
                               without removing it
      4 + @ @ ;                fetch the value pointed to by
                               headPtr

: #Que  ( <name> – n )        return number of values in the
                               queue
      @ ; MACRO

2048 CONSTANT bufMax          maximum buffer size in words
bufMax buildDQ DQ0
bufMax buildDQ DQ1
```

Circular Buffer Mode

Control start and stop with timer 0 interrupt enable/disable

```
VARIABLE readPtr              address of current buffer being
                              read from
VARIABLE writePtr             address of current buffer being
                              written to

: circBuf  ( – )              fetch value from current
                               circular buffer and put it in
                               the DAC scratchpad
      readPtr @               get address of current buffer
      Cbuf> >RH ;             get a value and put it in the
                               scratchpad

: stop  ( – )                 stop the output by turning off
      TIMER0 MASK ; MACRO      the time base

: stop?  ( – flg )            tst for timer 0 interrupt
      TIMER0 MASKED? ; MACRO   disabled
```

Ping Pong Buffer Operation

Read the current queue until it is empty. Then switch the queue pointer to the other queue and continue. Start/stop with timer 0 interrupt enable/disable.

```
: pingPong  ( – )             fetch a value from the current
                               buffer. If the read empties the
                               buffer, then swap read buffer
                               pointers
      readPtr @ DUP            get buffer pointer
      Que>                     fetch value and flag
      IF >RH                   If true then put new value in
                               scratchpad
        #Que                   get #values left in queue
```

```
          0= IF                 if zero then swap queue pointers
            readPtr @           which buffer are we using now?
            DQ0 =
            IF DQ1              if using 0 then switch to 1
            ELSE DQ0            else switch to 0
            THEN
          readPtr !
          THEN
        ELSE stop               otherwise the queues are empty.
        THEN ;

: setBufSiz  ( addr n — )     set size of circular buffer at
                               the given address to n words,
                               range 1 to bufMax actually done
                               by setting endPtr
        bufMax MIN 1 MAX      first check limits
        3 + CELLS             offset from start of the buffer
                               to end
        OVER +                add to start of buffer
        SWAP 6 + ! ;          and store in endPtr
```

Output Buffer Once and Stop When it is Empty

```
: onceOut  ( — )              fetch a value from the current
                               queue. If the queue is empty,
                               suspend output.
        readPtr @             get buffer pointer
        Que>                  and fetch a value
        IF >RH                if flag TRUE then output valid
                               point
        ELSE stop             otherwise the queue is empty
        THEN ;                mask off timer interrupts
GUARD

This routine is used for inputting data from the FIFO and
will be set up for vectored execution to allow alternate
input mechanism for demonstration.

: _readFifo  ( — n )          read the next value from the
                               input FIFO
        BEGIN                 need to wait if the FIFO is
                               empty
          fifoMT? NOT
        UNTIL
        fifo@ ;               and then fetch a value from it

VARIABLE 'readFifo            the execution address for
                               reading input

: readFifo 'readFifo @ EXECUTE ;

VARIABLE goAdr                holds the execution address for
                               the mode to fast sine mode

HEX 0FFF CONSTANT cntMsk DECIMAL  Block Size Mask
```

The Individual Input Command Routines

```
: fastSine  ( — )             operate in constant amplitude
        0 mode !               fast sine mode
        ['] nextPt goAdr !     execution address for fast sine
                               mode
        readFifo               read frequency from input queue
        setFreq ;              and set sine frequency

: accuSine  ( — )             accurate sine mode
        1 mode !               allows more accurate amplitude
                               generation, amplitude scaling,
                               and DC offset
        ['] newAccuPt goAdr ! execution address for accurate
                               sine mode
        readFifo               read frequency from input queue
        setFreq ;              and set sine frequency

: bufLoop0  ( — )             set up to loop on circular
                               buffer, buffer 0
        2 mode !
        ['] circBuf goAdr !    circular buffer execution
                               address
        DQ0 readPtr ! ;

: bufLoop1  ( — )             set up to loop on circular
                               buffer, buffer 1
        2 mode !
        ['] circBuf goAdr !    circular buffer execution
                               address
        DQ1 readPtr ! ;

: pingPong0  ( — )            set up to output from queue 0
        3 mode !               until it's empty and then from
                               queue 1 until empty
        ['] pingPong goAdr !   ping pong mode execution address
        DQ0 readPtr ! ;

: pingPong1  ( — )            set up to output from queue 1
        3 mode !               until it's empty and then from
                               queue 0 until empty
        ['] pingPong goAdr !   ping pong mode execution address
        DQ1 readPtr ! ;

: bufOut0  ( — )              set up to output from queue 0
        4 mode !               until it is empty
        ['] onceOut goAdr !    once out buffer execution
                               address
        DQ0 readPtr ! ;

: bufOut1  ( — )              set up to output from queue 1
        4 mode !               until it is empty
        ['] onceOut goAdr !    once out buffer execution address
        DQ1 readPtr ! ;

: start ( — )                 the start output command
        TIMER0 UNMASK          enable timebase interrupts
        BEGIN                  go into an infinite output loop
        RX@                    read stale data flag
        IF goAdr @ EXECUTE THEN  if set then output a new data
                               point
        stop?                  terminate if timer 0 stopped
        UNTIL ;

: setAmpl  ( — )              set amplitude scaling factor in
                               millivolts range 0..2000.  Will
                               be quantized to a multiple of 50
                               millivolts RMS.  Used only for
                               accurate sine mode
        readFifo               read the next input value
        0 MAX 2000 MIN         make sure it is within limits
        50 /                   and scale it
        scale ! ;

: setOffset  ( — )            set the offset that is
                               subtracted from the calculated
                               amplitude in accurate sine mode
        readFifo               get the value from the input
                               stream
        offset ! ;             and store it without any bounds
                               checking

: setFilt  ( — )              set the lowpass filter cutoff
                               frequency
        readFifo setCutoff ;

: getPeriod                    set sample period
        readFifo               read value from input stream
        setPeriod ;

: ldBuf                        load data into the queue
                               specified by the data words
        readFifo               get the parameter block to
                               determine which queue to load
                               and what size
        DUP 0<                 if MSB set then use buffer 1
                               else use 0
        IF DQ1 ELSE DQ0 THEN
        DUP writePtr !
        DUP clearQ             reinitialize the queue
        SWAP cntMsk AND        mask upper four bits of
                               parameter block ;
        TUCK setBufSiz         set buffer size
        1- FOR                 now set up a loop to load the
                               data in
          readFifo             get a value
```

```
        writePtr @ >Que     and put it in the queue
        IF                  >Que returns TRUE if queue is
          2DROP             full throw away the address &
          EXIT              data and exit
        THEN
      NEXT ;
```

Table of Execution Vectors for the Allowable Commands

```
table parse ' NOP ,           command 0 is a NOP
            ' fastSine ,      command 1
            ' accuSine ,      command 2
            ' setAmpl ,       command 3
            ' bufLoop0 ,      command 4
            ' bufLoop1 ,      command 5
            ' pingPong0 ,     command 6
            ' pingPong1 ,     command 7
            ' bufOut0 ,       command 8
            ' bufOut1 ,       command 9
            ' ldBuf ,         command 10
            ' setOffset ,     command 11
            ' setFilt ,       command 12
            ' getPeriod ,     command 13
            ' stop ,          command 14
            ' start ,         command 15

HEX
FFF0 CONSTANT cmdMsk

: doCmd  ( n - )            parse input word and perform the
                            command
      DUP cmdMsk AND        check that upper 12 bits are zero
      IF                    If not zero then it is an
        DROP                unrecognized command. Discard
        ." Unrecognized Command " CR
      ELSE parse EXECUTE    Otherwise, execute it via the
      THEN ;                parse table

DECIMAL
: initialize                initialization routine
      DISABLE               disable interrupts
      initTimers            set timer clock sources to all
                            internal
      intSetup              install timer 0 and timer 1
                            interrrupt routines
                            initialize variables
      0 phase ! 100 phaseInc !  initialize the phase
                            variables to some
      0 delPhase ! 0 delPhaseInc !  default values
      0 offset ! 40 scale ! initialize variables to some
                            value
      40000 cutoff !        initialize lowpass filter cutoff
      0 mode !              initialize mode
      1000 sineFreq !       sinewave frequency to 1000 Hz
      DQ0 readPtr ! DQ0 writePtr ! initialize buffer pointers
      ['] _readFifo 'readFifo !  pointer to read FIFO
                            executable
      SELCPR                all code page memory
      ENABLE ;              enable interrupts

: main  ( - )               the main high level loop
      initialize
      BEGIN
      readFifo              get a command from input stream
      doCmd                 and execute it
      AGAIN ;               an infinite loop
GUARD
```

Demonstration Code Source Listings

These are some sequences for demonstrating the operating modes.

```
DECIMAL
512 CONSTANT cqMax
cqMax buildCQ cmdQ          a queue for commands

: >cq  ( n - )   cmdQ >Que  used for loading a value into
                            the cmd queue
      IF 2DROP              the queue is full
        ." command queue full" CR
      THEN ;

: fixCQ  ( - )              used to adjust end pointer and
      cmdQ DUP #Que setBufSiz   tail pointer
      cmdQ DUP 4 + @ SWAP 2+ ! ;

: demoCircSq  ( - )         generate square wave 2V pk-pk,
      cmdQ clearQ           1 KHz using circular buffer mode
      cmdQ cqMax setBufSiz
      4 >cq                 buffer loop, buffer 0, command
      13 >cq 799 >cq        set period command, 100
                            microseconds
      12 >cq 40000 >cq      set filter cutoff to 40 KHz
      10 >cq                load buffer command
      20 >cq                buffer 0 20 values
      1 FOR
        4 FOR  10000 >cq NEXT  five points of +1V
        4 FOR -10000 >cq NEXT  five points of -1V
      NEXT
      15 >cq                start command
      fixCQ ;               adjust endPtr and tailPtr

: demoFsine20  ( - )        generate a 20 KHz sinewave, fast
      cmdQ clearQ           mode
      cmdQ cqMax setBufSiz
      1 >cq 20000 >cq       fast sine command, 20 KHz
      12 >cq 20000 >cq      set filter cutoff to 20 KHz
      15 >cq                start command
      fixCQ ;               adjust endPtr and tailPtr

: demoPsine1  ( - )         generate a 1 KHz sinewave, fast
      cmdQ clearQ           mode
      cmdQ cqMax setBufSiz
      1 >cq 1000 >cq        fast sine command, 1 KHz
      12 >cq 1000 >cq       set filter cutoff to 1 KHz
      15 >cq                start
      fixCQ ;               adjust endPtr and tailPtr

: demoAsine1  ( - )         generate a 1 KHz sine, accurate
                            mode
      cmdQ clearQ           90 mV offset, 1VRMS
      cmdQ cqMax setBufSiz
      2 >cq 1000 >cq        accurate sine mode, 1 KHz
      3 >cq 1000 >cq        set amplitude to 1000 mV
      11 >cq 900 >cq        set offset to 90.0 mV
      12 >cq 1000 >cq       set filter cutoff to 1 KHz
      15 >cq                start command
      fixCQ ;               adjust endPtr and tailPtr

: demoPp  ( - )             set up ping pong command queue
      cmdQ clearQ
      cmdQ cqMax setBufSiz
      6 >cq                 ping pong mode, buffer 0 start
      13 >cq 399 >cq        50 microseconds sample period
      12 >cq 40000 >cq      set filter cutoff to 40 KHz
      10 >cq 30 >cq         load buffer 0 with 30 points
      10 0 DO               the first 10 points
        I 40 * sine 2/ >cq  half sine cycle
      LOOP
      9 FOR 0 >cq NEXT      ten points of zero
      10 0 DO               ten points of another 1/2 cycle
        I 40 * sine 2/ 2/ >cq  half sine cycle
      LOOP
      10 >cq 21 -32768 OR >cq  load buffer 1 with 21 points
      9 FOR 0 >cq NEXT      ten points of zero
      11 0 DO
        I 80 * sine 2/ >cq  one cycle of 4 KHz
      LOOP
      15 >cq                start
      fixCQ ;               adjust endPtr and tailPtr

: demoOnce  ( - )           set up command queue for 1 cycle
      cmdQ clearQ           of linear attenuated 100 Hz sine
      cmdQ cqMax setBufSiz
      9 >cq                 once out from buffer 1 command
      13 >cq 199 >cq        25 microsecond sample period
      10 >cq 400 -32768 OR >cq  load buffer 1 with 400 points
      800 0 DO
        I sine 800 I - 800 */ >cq
      2 +LOOP
      15 >cq                start command
      fixCQ ;               adjust endPtr and tailPtr
```

```
: _rdCQ  ( – n )              return next value from command
       BEGIN cmdQ Que> UNTIL ;   queue

: cmdQMT?  ( – flg )          return TRUE if cmd queue empty
       cmdQ #Que 0= ;

: demoInit  ( – )             initialize for demonstration
       initialize              mode
       ['] _rdCQ 'readFifo ! ;  revector read FIFO command

: demoMain  ( – )             demo execution loop similar to
                               application main. The only
                               difference is that this loop
                               isn't an infinite loop
       demoInit
       BEGIN
         readFifo             get a command
         doCmd                and execute it
         cmdQMT?              loop if there is another command
       UNTIL  ;

: dMenu  ( – )                  build choice menu
     CR ." Choose a demo waveform"
     CR ."  a - 1KHz square wave using circular buffer"
     CR ."  b - 20KHz sinewave fast sine mode"
     CR ."  c - 1KHz sinewave fast sine mode"
     CR ."  d - 1KHz sinewave accu sine mode"
     CR ."  e - ping pong mode"
     CR ."  f - 100Hz attenuated sine once out mode"
     CR ."  q - quit"
     CR ." Choice: " ;

: choose  ( – n )             return a value 0..5 based on
menu choice
       dMenu
       BEGIN
         KEY DUP ASCII q = IF DROP QUIT THEN
         ASCII a - DUP 0 6 WITHIN
       NOT WHILE
         DROP
         ."   error."
         CR ." Choice: "
       REPEAT CR ;

table dVect                    a table of queue setups
       ' demoCircSq ,
       ' demoFsine20 ,
       ' demoFsine1 ,
       ' demoAsine1 ,
       ' demoPp ,
       ' demoOnce ,

: demo  ( – )                  start up a demo
       BEGIN
         choose dVect EXECUTE
         demoMain
       AGAIN ;

: shoSparms CR ." phase: " phase @ .
            CR ." increment: " phaseInc @ .
            CR ." delta phase: " delPhase @ .
            CR ." delta increment: " delPhaseInc @ .
            CR ." frequency: " sineFreq @ U.
            CR ." offset: " offset @ .
            CR ." scale: " scale @ . ;
: shoParms CR ." mode: " mode @ DUP .
       0 2 WITHIN IF shoSparms THEN
       CR ." cutoff: " cutoff @ U.
       CR ." timebase period: " tbPeriod @ 1+ U. ;
```

Some Performance Testing

```
VARIABLE high
VARIABLE low

: tst  NOP newPt NOP
       BEGIN RX@ IF goAdr @ EXECUTE THEN
       stop? UNTIL ;

: atst 1 high ! 10000 low !
       stop
       9999 FOR
       TC2@ tst TC2@ -
       DUP low @ UMIN low ! high @ UMAX high !
       NEXT
       CR ." max " high @ .
       CR ." min " low @ . CR ;

: btst 1 high ! 10000 low !
       stop
       39 FOR
       TC2@ tst TC2@ -
       DUP low @ UMIN low ! high @ UMAX high !
       NEXT
       CR ." max " high @ .
       CR ." min " low @ . CR ;
```

Editor, from page 36

been a resounding success. I am happy to report several outstanding articles in the wings, including X10 home control, work with the NZCOM virtual BIOS, and a discussion of CRC in Forth and telecommunications protocols. A wirewrapped universal SCSI adapter will be an upcoming topic. A particularly bright note concerns a new column you will find in this issue: Paul Chidley brings us to *Hardware Heaven*.

Our readers are our authors, our critics and our collective editors. I have never worked for a finer group of people!

We have two new services. One is specifically to make life easier for overseas subscribers: No longer must you have a bank draft in US funds cut to renew or purchase back issues. We now accept seven different charge cards! Use your Visa, MasterCard, Discover, Carte Blanche, Diner's Club, JCB and EuroCard instead.

US and Canadian readers can use the cards, of course, plus our new 24-hour tollfree telephone number. Call us at 800-424-8825. I'll have an answering machine on if we aren't available.

Take Some Credit

TCJ's growth is a joint effort. You, the readers and our authors have brought many of our new readers. Thank you! This is what a grassroots effort is all about—working together gives us a resource, in this case a magazine, that we couldn't have alone. I have recognized the value of word-of-mouth advertising since the beginning. That is why we have the "Sponsor a Friend" program. When a friend or colleague of yours enters a subscription as a result of your discussions, you should get something for it. I'll add a free issue to your subscription when the friend sends payment with the order and identifies you as the sponsor.

See Editor, page 44

Editor, from page 43

It bothers me to see new people coming in and I don't know who sent them! How can I reward you?

Addresses, Back Issues and Such

Ever hear that if an infinite number of monkeys were given infinite time to pound on typewriters that they would eventually write a best selling novel? Monkeys are intelligent creatures. I hope one day to be their equal.

A handful of invoices are returned every month with address corrections. This bothers me; these folks aren't getting their copies. Sure enough, the requests to remail "lost" copies soon follow. I want to help, friends, but this gets real expensive, right fast. So I gave it infinite effort and have a solution. It comes in two steps:

1) For last issue and this, I have put the notice ADDRESS CORRECTION REQUESTED on the mailing cover. For this, the post office "sells" me the new addresses. (Yes, we pay for this). This will get the stray addresses in line.

2) Once this is done, I will add FORWARDING POSTAGE GUARANTEED. The post office will forward the issue to you and ask you to pay. It will cost US subscribers about $1.75 a copy. If you refuse it, I will pay your $1.75 and then another $1.75 when the issue is returned, as well as the address correction.

This seems as far as I should go to ask for folks to send in address changes. Any replacements will then be treated as back issue sales. Does this sound reasonable?

Speaking of back issues: Ester warns me that we are just about sold out of several issues. I don't have the list in front of me, but we have already dropped issues 8 and 19. They are no longer available. In time I hope to have the ability to reprint sold out issues, but don't count of it any time soon. Just thought I'd be fair and let you know.

Meanwhile, you will get a refund check if you order an issue that has sold out.

The Chicken Heart that Ate New York City

I remember one of Bill Cosby's early comedy albums when he had a skit about listening to scary radio shows as a child and frightening himself silly. In one, there was *The Chicken Heart that Ate New York City*. I am here to report that I met that Chicken Heart and He Is Me. Though I am on the way back down, I put on an extra twentyfive pounds since we last communicated. How?

I quit smoking!

Yeah, big deal. Nothing to tell the world about, certainly not related to computers. But it's a personal victory. I have chewed my way through the Mid-Atlantic states and eaten half of New England. New Hampshire no longer exists. I gained control of my appetite somewhere in southern Vermont. But I have been off cigarettes since November. And I feel great.

I'll make a deal with you. I will give two free issues to any *TCJ* subscriber who quits smoking and goes six months without a cigarette. We'll do this on the honor system. You tell me you did smoke and you don't any longer. Tell me when you quit and how long you smoked. I'll add the issues to your subscription. Guess we need an "expiration date" for this. If you quit now, six months would be the end of August. So, the offer expires the end of September, 1992. Sound good?

And I will make another deal. The first *TCJ* reader who sees me smoking a cigarette will have a free lifetime subscription. No expiration on this deal!

Well, enough of this. It's time. Enjoy yourself.●

Reader, from page 2

seems to be sold by a number of Radio Shack Color Computer places, so you can find it there as well. The interface is quite simple and it should be possible to hook it to almost anything; I'm currently planning connections to one of my PDP-11s and an rtVAX.

R.I., Logan UT

The device discussed in last issue was not the Emerald Microware unit. Ampro had sold a true SCSI daughter board, and George Warner tells me he still wirewraps them for friends. I am hoping George will tell us what he does.

By the way, information gives 503-641-8088 as the number to reach Emerald Microware. They are in Aloha, OR.—Ed.

I was happy to receive issue 52, especially the articles about the Yasbec and the CPU280. I'm working towards a "reinvention" of the CPU280 but with a SCSI interface instead of the floppy.

M.D-S, Holte Denmark

I understand that Tilmann Reh and Uwe Herczeg are cooking up an IDE interface for the CPU280. For myself, I'd rather the SCSI.—Ed.

I have a suggestion concerning typesetting in *TCJ*. It is most important to have every article on contiguous pages. This is the main rule of making the articles readable and should have the absolutely highest priority. Every other rule has to be considered only after this main rule is already true. The easiest way of grouping the articles would be to simply add one after another. This would sometimes cause articles to start in the middle of a page but that wouldn't bother me.

If you have a rule saying that articles have to start at the top of a page, then you could achieve this without injuring the main rule: just put the rest of the previous article (which in this case will always be less than one page) on the bottom of the new page and the next article at the top.

T.R.,Siegen Germany

I've taken some heat for continuing articles on other pages but accepted this as the cost of providing as much information in each issue as we have space for. This is, after all, the way most other magazines in the United States are laid out. Your suggestion seems to come from the other extreme: lay the journal out as if it were a letter. Friends from Europe tell me this is standard practice in Germany so I understand. The readers in this country would not be familiar with such a layout, however.

I will keep the articles together as much as possible but will put the tailends where we can make best use of the remaining pages. Meanwhile, I will use this and my own column to fill the small voids as much as I can to leave the features more intact.—Ed.

Is Wayne Masters still around? I noticed in the RCP/M lists that his Potpourri BBS isn't listed and I am having a bit of a problem trying to get BYE on an Altos 8000 to work correctly with an AMT 2400 modem.

Good question. Anyone with an answer?—Ed.●

The Development of TDOS

By Guy Cousineau

COLECO introduced the Adam computer several years back. At the time, it was one of the great buys in the home computer arena and offered CP/M 2.2. Unfortunately, it came with digital tape drives instead of disks. A disk operating system such as CP/M was unfriendly at best. I had a look and asked myself, "What can I do with this?" I shelved it.

Months later, Tony Morehen, a long time friend, bought an Adam and became interested in CP/M. He worked on it from tape for a while. During this time, I was still working in BASIC. Once disk drives became available, Tony started working in improving the Adam implementation of CP/M. The BIOS was not flexible enough to handle all the devices that were, by then, becoming available. He wrote a patch to support a parallel printer and another for disks. Other patches fixed bugs in the allocation vector sizes for the expansion RAM disk. There were patches for 80 column screens; previously we used a 32 column television screen.

My interest in CP/M grew. I learned Z80 code by disassembling and fixing the Adam's BASIC. CP/M modem programs became available and we began acquiring software. One of the more useful was a Z80 assembler.

Tony decided we would rewrite the CCP resulting in a 2K CCP. Among other things, it included a sorted directory function, and built-in copy command supporting user areas.

After our success with the CCP, we turned to the BDOS. Tony looked at P2DOS, Super BDOS and others. He found merits in each but we felt we could do better. Basing ourselves on these systems, we developed a BDOS with time stamping and automatic disk logging. That first crack at the BDOS was relatively easy.

Now we were playing with power.

We turned to the big task: the BIOS. Working on a combination 32/80 column system, we rewrote the entire BIOS. We speed up the disk drivers and added support for four different formats. With the entire system in source code, we were finally able to get away from the SYSGEN, SAVE, DDT, LOAD, GO sequence which made patch installations so tedious. We could append a SYSGEN function to the code which would install the entire system in one pass. By this time, we discovered the Video Display Processor was capable of a 40 column display. I started work on patching that support code in. Eventually, we dropped the 32 column mode entirely.

Development did not stop there. Tony wrote an installation program which would prompt for disk drive sizes, serial port settings, default colours, macro key configurations, I/O byte settings and the like prior to installing the system. The user does not need to install patches at all. When system configuration changed, they only needed to reinstall the system.

SUBMIT was another thorn in our side. It was slow! We worked on a new type of Submit which worked like MSDOS Batch files. Support for 32 user areas became a necessity when we developed support for hard disk drives. We added named directories as well. Finally, we made the installation program smarter by scanning the network and configuring the system based on the devices we found.

Incompatibilities arose: certain drive specifications, hard drive interfaces, clock chips and such gave trouble. Terminal installation also became a problem. Inexperienced users were having difficulty installing programs for all the different terminals available. We developed a standard header for all TDOS programs. Each user requires one terminal overlay which he can MLOAD to all our programs. We extended this approach to our generic CP/M work as well, which made it quite easy for the average person to install our programs.

TDOS was now available in several versions and comes with a complete set of utilities:

- Change file user area without copying
- Named directories
- Sort directories with directory name first to speed Change Directory command
- Prepare directory for date stamping
- Format disks with sysgen and clear directory options
- Set date (for systems without clocks)
- I/O Byte setting (replaces STAT)
- Logical disk size changes to temporarily read other logical formats
- Set search path
- Time/Date directory with command line date mask

These and other utilities, along with the TDOS system, replace all the utilities which came with CP/M 2.2. and help get the most productivity out of CP/M. Some of the latest TDOS development include ZCPR. A few people use it, but I find that with what we have in TDOS right now, we don't need to go to ZCPR unless we want to use some of its more advanced features. Our next project is to go to CP/M Plus emulation by introducing bank switching.

The nice thing about TDOS is that it is *free*. We would have done it just for ourselves anyway, by why not share it. Developing TDOS taught me a great deal about machine language programming. I have become an expert on writing fast and compact code. Now that I have the CP/M bug, I don't know why I would ever want to change operating systems.●

Guy Cousineau represents the present day CP/Mer. Not content to use a system written by others, he has joined with others to delve deeply into the inner workings of his computer to vastly expand its capabilities. He offers the fruits of his labors while gaining intimate knowledge of micro processors. Guy may be contacted at 1059 Hindley Street, Ottawa K2B 5L9, Canada.

TCJ The Computer Journal
The Spirit of the Individual Made This Industry

Back Issues

Telephone Orders: (800) 424-8825 / (908) 755-6186, 24 hours

Special Close Out Sale
Issues 1, 2, 3, 4, 8 only

3 or more, $1.50 each postpaid in the US
$3.00 postpaid airmail outside US.

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

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

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

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

Issue Number 8: [illegible]
Sold Out

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

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

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

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

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

Issue Number 24:
- Selecting & Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assemble Code for CP/M
- The C Column: Software Text Filters
- Ampro 186 Column: Installing MS-DOS Software
- The Z-Column
- NEW-DOS: The CCP Internal Commands
- ZTime-1: A Real Time Clock for the Ampro Z-80 Little Board

Issue Number 25:
- Repairing & Modifying Printed Circuits
- Z-Com vs. Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro LB
- Building a SCSI Adapter
- NEW-DOS: CCP Internal Commands, Pt. 1
- Ampro 186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:
- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside Ampro Computers
- NEW-DOS: CCP Internal Commands, Pt 2
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

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

Issue Number 28:
- Starting Your Own BBS
- Build an A/D Converter for the Ampro Little Board
- HD64180: Setting the Wait States & RAM Refresh using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:
- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM Disk to Ampro Little Board, Part 1
- Using the Hitachi hd64180: Embedded Processor Design
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:
- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro Little Board
- 3200 Hackers' Language
- MDISK: Adding a 1 Meg RAM Disk to Ampro Little Board, Part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:
- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk Parameters & their variations
- XBIOS: A Replacement BIOS for the SB180
- K-OS ONE and the SAGE: Demystifying Operating Systems
- Remote: Designing a Remote System Program
- The ZCPR3 Corner: ARUNZ Documentation

Issue Number 32:
- Language Development: Automatic Generation of Parsers for Interactive Systems
- Designing Operating Systems: A ROM based OS for the Z81
- Advanced CP/M: Boosting Performance
- Systematic Elimination of MS-DOS Files: Part 1, Deleting Root Directories & an In-Depth Look at the FCB
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII Terminal Based Systems
- K-OS ONE and the SAGE: System Layout and Hardware Configuration
- The ZCPR3 Corner: NZCOM and ZCPR34

Issue Number 33:
- Data File Conversion: Writing a Filter to Convert Foreign File Formats
- Advanced CP/M: ZCPR3PLUS & How to Write Self Relocating Code
- DataBase: The First in a Series on Data Bases and Information Processing
- SCSI for the S-100 Bus: Another Example of SCSI's Versatility
- A Mouse on any Hardware: Implementing the Mouse on a Z80 System
- Systematic Elimination of MS-DOS Files: Part 2, Subdirectories & Extended DOS Services
- ZCPR3 Corner: ARUNZ Shells & Patching WordStar 4.0

Issue Number 34:
- Developing a File Encryption System.
- Database: A continuation of the data base primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking executive.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.
- The Computer Corner

Issue Number 35:
- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8088 software to produce modifiable assembly source code.
- Real Computing: The NS32032.
- S-100: EPROM Burner project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System. Part 1: Selecting your assembler, linker and debugger.
- The Computer Corner

Issue Number 36:
- Information Engineering: Introduction.
- Modula-2: A list of reference books.
- Temperature Measurement & Control: Agricultural computer application.
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE.
- Real Computing: NS32032 hardware for experimenter, CPUs in series, software options.
- SPRINT: A review.
- REL-Style Assembly Language for CP/M & ZSystems, part 2.
- Advanced CP/M: Environmental programming
- The Computer Corner.

Issue Number 37:
- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers.
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER.
- Information Engineering: Basic Concepts: fields, field definition, client worksheets.
- Shells: Using ZCPR3 named shell variables to store date variables.
- Resident Programs: A detailed look at TSRs & how they can lead to chaos.
- Advanced CP/M: Raw & cooked console I/O
- Real Computing: The NS 32000.
- ZSDOS: Anatomy of an Operating System: Part 1.
- The Computer Corner.

Issue Number 38:
- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS: Anatomy of an Operating System, Pt 2.

Issue Number 39:
- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:
- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

Issue Number 41:
- Forth Column: ADTs, Object Oriented Concepts.
- Improving the Ampro LB: Overcoming the 88Mb hard drive limit.
- How to add Data Structures in Forth
- Advanced CP/M: CP/M is hacker's haven, and Z-System Command Scheduler.
- The Z-System Corner: Extended Multiple Command Line, and aliases.
- Programming disk & printer functions w/C.
- LINKPRL: Making RSXes easy.
- SCOPY: Copying a series of unrelated files.
- The Computer Corner.

The Computer Corner

By Bill Kibler

Well another month has gone by, with revelations galore. For myself, I have finally sold one house that has been on the market for over a year. Hopefully I can get more work done now. I have also been checking out lots of software and even went to one course last month. I have played with DR DOS 6.0 and have something to say about version control software (the course I attended).

What Works

I tried to load DR DOS and found some problems. Our company has been checking out other network systems. For those who didn't see the trade journals, Novell Netware bought Digital Research Inc. which gave them the rights to DR DOS. You may remember DRI as the creators of CP/M. From reading a few of the "help me" files, I can see what the purchase of DRI really means to Novell. With DR DOS, Novell no longer has to deal with all the bugs and uppity dealings of Microsoft.

All this comes about from the many companies having trouble with Microsoft and their shifting between OS2 and MSDOS. What also helped this along was Novell needing a stable DOS environment as well as access to the source code. DR DOS version 6.0 came with Novell's new NETLITE program. NEWLITE is supposed to be able to provide a server-less LAN network. We have just started testing the product at work so more later.

DR DOS 6.0 has all the MSDOS features plus some. After all, CP/M was the basis for MSDOS and most of the early system calls were the same. I tried their SID, which replaces DEBUG, and would much rather use it any day. It even has a built in list of options. I always hate using DEBUG because I will forget some syntax and have to hunt down a book to complete my work. However their DOSBOOK program (a system help manual) has an annoying problem. Hitting the ESC key will put you at the main menu and not back at the sub menu you just came from. With several hundred topics to view, jumping back to main is most troublesome.

Looking at some of their utilities you can see how they will be able to provide many new features because they have control of the DOS source code. One of those features is a multi-tasking like utility that I didn't get to try, yet! a

aThis makes me feel that any company doing LAN based work needs to drop the other formats and move to Novell. Doing so will kill two birds with one program, cut your LAN problems in half and give you a better DOS.

VCS

What is VCS you might ask? VCS stands for Version Control Software. That means an automated means of tracking changes in your software. Our programs have been on a mainframe and as such have absolutely no method of tracking who did what to where. We have 6 and 7 year old files that appear to have been changed, but the object module that was being used was from before the changes. The question then is what did happen and were the changes really needed or not.

The product I have been sent to learn is Intersolv's Poly VCS. There are many products from an assortment of vendors that do the same thing and I would recommend you check them all out before trying or buying one. PVCS (Poly VCS) is being used elsewhere in our company and so we too went with it. I feel that all of the available programs will work the same, some may be better for your personal environment, however.

PVCS, like most of the VCS programs, allows files to be checked out of the hard drive and locked (read only so old copy can't be written to). You then make your changes and tests on your local drive. When you are sure it is all correct, you then log it back into the server. At that point you are prompted for a description of your changes and that is recorded in the log file. The log file contains all the source code and all the changes as well. The changes are short sections of differences and PVCS calls them DELTAS. By storing just the changes (deltas) and not a second or third copy of the file, it is possible to save hundreds of histories with a marginally larger file. This also allows you to re-create or check out older or past versions of releases if needed.

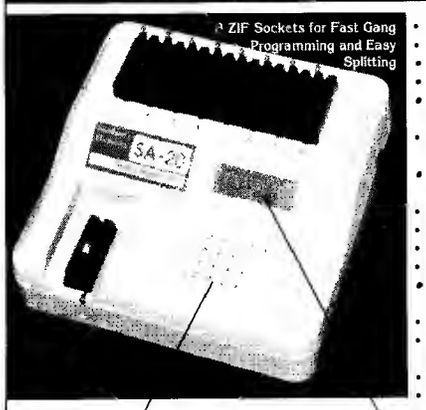
We have many past releases of our products in which we have no way of re-creating them. There also is no way of knowing if past bugs have in fact been fixed or in which release version they were fixed. PVCS and its log and tracking system will do all that for us. It will also print out a list of who tried to do what on the system. That means you have full security if you want, complete with tracking who is trying to break the system. Security can be set at the file or group of files level. You can prevent programmers from getting outside their area of responsibility if that is a problem.

In our case using PVCS is somewhat overkill for our project. It is an older product that is slowing down as far as the changes are concerned. Also we will only have about five people using it and so a system that can handle hundreds of users is a bit overkill. Here again is a product in which the book will explain it all, but a course will cut the learning curve to about 20 percent of the time. I worked with their MAKE program (automated assembly and linking program) and had problems finding all the fine print. I mean fine only from the point of brief and well hidden in some part of the

See Computer Corner, page 23

EPROM PROGRAMMERS

Stand-Alone Gang Programmer $750.00



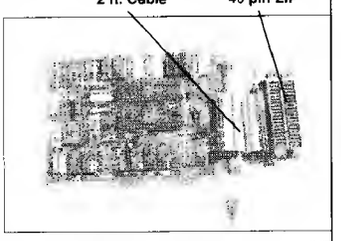
- Completely stand-alone or PC driven
- Programs E(E)PROMs
- ***1 Megabit of ORAM***
- ***User upgradable to 32 Megabit***
- ***.3/.6" ZIF socket, RS-232, Parallel In and Out***
- 32K internal Flash EEPROM for easy firmware upgrades
- ***Quick Pulse Algorithm (27256 in 5 sec, 1 Megabit in 17 sec.)***
- 2 year warranty
- Made in U.S.A.
- Technical support by phone
- Complete manual and schematic
- ***Single Socket Programmer also available. $550.00***
- Split and Shuffle 16 & 32 bit
- 100 User Definable Macros, 10 User Definable Configurations
- Intelligent Identifier
- Binary, Intel Hex, and Motorola S

20 Key Tactile Keypad (not membrane) **20 x 4 Line LCD Display**

Internal Programmer for PC $139.95

New Intelligent Averaging Algorithm. Programs 64A in 10 sec., 256 in 1 min., 1 Meg (27010, 011) in 2 min. 45 sec., 2 Meg (27C2001) in 5 min. Internal card with external 40 pin ZIF.

2 ft. Cable **40 pin ZIF**



- Reads, verifies, and programs 2716, 32, 32A, 64, 64A, 128, 128A, 256, 512, 513, 010, 011, 301, 27C2001, MCM 68764, 2532
- ***Automatically sets programming voltage***
- Load and save buffer to disk
- Binary, Intel Hex, and Motorola S formats
- ***Upgradable to 32 Meg EPROMs***
- ***No personality modules required***
- 1 year warranty • 10 day money back guarantee
- Adapters available for 8748, 49, 51, 751, 52, 55, TMS 7742, 27210, 57C1024, and memory cards
- Made in U.S.A.

NEEDHAM'S ELECTRONICS
4539 Orange Grove Ave. • Sacramento, CA 95841
Mon. - Fri. 8am - 5pm PST

Call for more information
(916) 924-8037
FAX (916) 972-9960

C.O.D.

Cross-Assemblers as low as $50.00
Simulators as low as $100.00
Cross-Disassemblers as low as $100.00
Developer Packages
as low as $200.00(a $50.00 Savings)

A New Project
Our line of macro Cross-assemblers are easy to use and full featured, including conditional assembly and unlimited include files.

Get It To Market--FAST
Don't wait until the hardware is finished to debug your software. Our Simulators can test your program logic before the hardware is built.

No Source!
A minor glitch has shown up in the firmware, and you can't find the original source program. Our line of disassemblers can help you re-create the original assembly language source.

Set To Go
Buy our developer package and the next time your boss says "Get to work.", you'll be ready for anything.

Quality Solutions
PseudoCorp has been providing quality solutions for microprocessor problems since 1985.

BROAD RANGE OF SUPPORT

- Currently we support the following microprocessor families (with more in development):

Intel 8048	RCA 1802,05	Intel 8051	Intel 8096
Motorola 6800	Motorola 6801	Motorola 68HC11	Motorola 6805
Hitachi 6301	Motorola 6809	MOS Tech 6502	WDC 65C02
Rockwell 65C02	Intel 8080,85	Zilog Z80	NSC 800
Hitachi HD64180	Motorola 68000,8	Motorola 68010	Intel 80C196

- All products require an IBM PC or compatible.

So What Are You Waiting For? Call us:

PseudoCorp
Professional Development Products Group
716 Thimble Shoals Blvd, Suite E
Newport News, VA 23606
(804) 873-1947 **FAX: (804)873-2154**

William P Woodall • Software Specialist

Custom Software Solutions for Industry:

Industrial Controls	Hardware Interfacing
Operating Systems	Proprietary Languages
Image Processing	Component Lists

Custom Software Solutions for Business:

Order Entry	Point-of-Sale
Warehouse Automation	Accounting Systems
Inventory Control	Local Area Networks
Wide Area Networks	Telecommunications

Publishing Services:

Desktop Systems	Format Conversions
Books	Directories
CBT	Interactive Video

33 North Doughty Ave, Somerville, NJ 08876 • (908) 526-5980