ZCPR 3.3 User's Guide

Written by
Jay P. Sage

Echelon, Inc.
885 N. San Antonio Road
Los Altos, CA 94022 USA
415/948-3820

Printed in the United States of America
First Printing: May 1987
Second Edition, First Printing: June 1987

9 8 7 6 5 4 3

Publisher's Foreword
15 May 1987

ZCPR 3.3 continues the traditions of convenience, quality, and achievement which are the hallmark of ZCPR. As with other ZCPR's, ZCPR 3.3 gives the user unparalleled power and capabilities, with the subsequent benefit of personal growth while learning to take complete advantage of the system. As a consequence of this growth and learning the user is changed by this software and its concepts, unlike most commercial software which seeks to appeal to the lowest common denominator amongst users.

We believe ZCPR 3.3 to be the most powerful software for controlling your computer system available for any machine or from any source. Yet, ZCPR continues to be enhanced, new concepts added, additional power put into the user's hands. Expect this process of growth to continue, as life does.

This publication is written as a supplement to Richard Conn's book ZCPR3: The Manual. The newcomer to ZCPR3 will find both publications necessary to a complete understanding of the system; previous users of ZCPR3 should find this document an adequate guide.

Good luck in your journeys with ZCPR3.

T A B L E O F C O N T E N T S

1. INTRODUCTION

This user guide describes version 3.3 of the ZCPR command processor, an upgrade from Richard Conn's original release of ZCPR3, version 3.0. Since we refer to these two versions so often in this document, we use an abbreviated notation. Z30 is used for ZCPR version 3.0, Z33 is used for ZCPR version 3.3, and ZCPR3 is used to refer to both versions generically.

The guide is divided into four main sections: Introduction, Command Processing Functions, Resident Commands, and Installation. In the introduction we first describe the design goals that guided the development of Z33. Then we give a brief summary of the features and benefits offered by Z33. Finally, we try to put Z33 in perspective by presenting a short account of the history of ZCPR from its inception in 1981. This historical account includes acknowledgments to a number of individuals who made particularly significant contributions to the development of version 3.3.

The command processor in CP/M has traditionally performed two distinct functions. One of them, as the name implies, is command processing -- accepting, interpreting, and loading user commands. The other has been the secondary role of providing the actual code for a number of basic command functions. In section 2 we describe the command processing services provided by Z33, and in section 3 we cover the resident commands that can optionally be included in the processor.

In the last section of the user guide, we describe the procedure for installing Z33, either in a conventional manually installed system or in one of Echelon's ingenious auto-install implementations, Z-Com or Z3-Dot-Com.

1.1. ZCPR33 Design Goals

In developing Z33, we have tried to achieve three qualities: compatibility, flexibility, and reliability. To allow programmers to make maximum use of system capabilities, we have also tried to provide as many information hooks as possible.

1.1.1. Compatibility

To the greatest extent reasonable, Z33 is compatible with Z30. No change is required to any part of the operating system other than the command processor itself. The memory allocation and system modules that you used with Z30 work with Z33. Thus, installing Z33 requires only that the new code be assembled and installed in place of the old command processor code.

The feel of Z33 to the user is basically the same, and almost all programs that ran under Z30 run identically under Z33. In a small number of cases programs run slightly differently under unusual conditions, such as when run from a ZEX script. In most of these cases, the program now runs correctly when before it did not. For example, when an error handler is invoked because of a command error that occurs in a ZEX script, input redirection is turned off automatically. Thus the user can decide how the error should be dealt

with. Under Z30, quite bizarre behavior would result under these circumstances, because many of the error handlers were not designed with ZEX in mind.

Two known changes have been made that cause Z30 versions of some programs no longer to work. One such change is in the way submit processing is carried out. Under Z30, the $$$.SUB file was kept on drive A in the user number that was current when the batch job was started. As a result, submit jobs could not change user areas during operation. In Z33 submit processing references user area 0 at all times. Consequently, the old submit utility, SUB.COM, does not work with Z33 except in user area 0. However, a new Z33 version, SUB33.COM, is already available.

The second change is in the way one of the flags in the message buffer is used. The error-handler flag at offset 0 in the message buffer formerly indicated whether or not an error handling command line had been installed by the user. This flag was redundant, since the information it provided could just as readily be determined from the error command line itself. The error-handler flag has been given a new, more important purpose in Z33, namely, to indicate the precise type of error that occurred. Because of this change, some programs, such as SHOW.COM, may not indicate properly whether or not an error handler is loaded. Corrected versions of affected programs will be available at the time Z33 is released or shortly thereafter.

1.1.2. Flexibility

The second design goal was to achieve simultaneously high degrees of flexibility, user control, and ease of use. Many features of ZCPR3 -- such as automatic searching for program files, extended command processing, and error handling -- are extremely convenient, but they come at the expense of speed. In Z33 we have maintained all the convenience features of Z30 (and even added some more), but we have given the user ways to impose greater control over the behavior of the system when he wishes to exercise it.

For example, when a command is prefixed with a space or a '/', extended command processing is invoked immediately, completely bypassing the usual search for a COM file. On the other hand, prefixing a command with an explicit directory reference forces the command processor to look only for a COM file (resident commands are bypassed) of that name and to look for it only in the specified directory; both the command search and extended command processing are disabled.

One of the hallmarks of ZCPR3 has been its ease of use. Unfortunately, this only applied to open (non-secure) versions. As soon as Z30 was configured in a secure mode, such as when it was used for remote access systems, much of this ease of use went out the window. The ability to refer to a directory area by name was a convenient option available to the user in an open system, but in a secure system it often became a burdensome requirement. A secure Z33 system need be no less convenient and easy to use than an open system. In fact, we use the same version of Z33 on our personal systems as we do on our remote access Z-Node systems. One way this is achieved is through the option to skip password checking on named directories when the wheel byte is set. Automatic conversion of DU (drive/user) references to equivalent named-directory references gives the user the freedom to employ whichever form is most convenient.

1.1.3. Reliability

The third design goal has been reliability, in two senses. First of all, we have tried to make the code rigorous, so that improper commands cannot impair the system. In Z30, for example, when the command tail was copied from the command line buffer to the command tail buffer at 80H, no checking was performed to prevent a tail longer than 128 bytes from overwriting the program loaded at 100H (the program is loaded before the tail is copied). This deficiency in Z30 could result in a fatal system crash or worse.

Many other reliability problems have been corrected. Here are three more examples. First, care has been taken to ensure that the DMA buffer is always set to the default location of 80H when disk directories are being scanned. With the GET command in Z30, for example, the load checking code would complain about an attempt to load a file over the operating system. Although loading of the file would then be aborted, memory had already been clobbered when the directory sector was read. Second, when programs are loaded into memory under Z33, overwrite protection is provided not only for the command processor but also for any resident system extension code (such as ZEX) that has been installed below the command processor. Third, command lines read in from submit files cannot write beyond the end of the command line buffer and damage other system modules in memory.

We have also tried to make Z33 more reliable from the point of view of providing far greater opportunity for recovery from user errors. Mistakes in the use of command-processor-resident commands now invoke the error handling facility, and a mechanism has been provided that allows non-CPR commands (either COM programs or RCP/FCP-resident commands) to engage error handling as well. As the multiple command line facilities of ZCPR3 are used more and more by automatic command line generators (aliases, shells, ZEX, and SUBMIT), it becomes more and more important to allow the user to intervene when a problem arises that prevents the proper operation of one command in a long sequence of commands. In addition to trapping far more errors, Z33 also provides much more information about the nature of the error and what may have caused it.

1.1.4. Information Hooks

We have tried to make available to programs as much information about the system as possible. This has been accomplished in several ways. First, additional information has been included in standard channels. Here are two examples.

The external file control block now contains not only the name of the command that was requested. It also includes the drive and user number where the command was actually located by the command processor during its search. The drive always has an explicit value in the range 1 to 16 if the program was a transient. A value of zero indicates that the command was a resident command.

When the file control blocks are parsed by the CPR parser, information is included that allows a program to determine 1) whether the directory specification was valid and 2) whether the directory specification was explicit (as opposed to the default). If an explicit directory (valid or invalid) was included, the drive byte in the FCB is filled in with a value in the range 1 to 16. If no explicit directory was given, the drive byte will be

zero. When a specified directory is invalid, because it either does not exist or is out of range, the current directory is substituted, but the record count byte (offset 15 in the FCB) is set to FFH instead of 0. In this way, a program can detect that an invalid directory and not the default directory was specified.

Some new information channels have also been provided by moving information that used to be kept internal to the command processor out to the message buffer. The currently logged drive and user number are kept there now. As a result, a program can change the logged-in directory without having to force a warmboot. This code also made it possible to log into user areas above 15, though this is risky because many programs are not prepared to find themselves in those high user numbers. Consequently, this feature is optional and not recommended for the general user. The submit-running flag has also been moved to the message buffer, so a program can now tell if a submit job is currently in operation, and an XSUB flag has been provided for controlling input redirection during submit jobs. Finally, several option bytes have been included at the beginning of the command processor code so that programs can determine what features are implemented.

1.2. Summary of Benefits

The new features of Z33 are described in detail in other sections of this manual. Here we describe the benefits you can gain from using ZCPR3 in general, and the benefits of using ZCPR 3.3 as compared to ZCPR 3.0. First, the benefits of using ZCPR3 over using ordinary CP/M.

1.2.1. Benefits over CP/M

- Disk user areas can be used easily to organize your files. ZCPR3 has extensive improvements over CP/M in this area, among which are: indication of current disk and user in the system prompt, use of the DU: (disk/user) expression to move between user areas, provisions for assigning meaningful names to given disk/user areas, and restricting access to given disk/user areas by unauthorized persons.

- ZCPR3 provides an extended system environment definition that defines your particular computer system in significant detail, and programs running under a ZCPR3 environment can be more intelligent about the specifics of your system in a fashion impossible under CP/M. For example, such programs can know all about the control sequences needed to make extensive use of your video display, can determine how fast your microprocessor runs, can know how many columns per line your printer supports, and much more. This means that the same exact program works with radically different screen displays or other hardware, without cumbersome patching or installation programs.

- Enhanced command processing makes your system much more efficient and responsive to your needs. Elements of this concept include multiple command handling, where ZCPR3 allows you to enter more than a single command at a time, or the PATH concept, which permits the programs you invoke to be automatically searched for in different user areas or disks from wherever you happen to be logged in. Another element is the SHELL feature, where complete turnkey menu-driven environments, command history recall and editing, or other user interfaces can be added to the system dynamically. The ALIAS feature allows the user to add new commands to his system and call them by names of

his choosing. An extended command processor can be added which allows the user to store hundreds of commands or programs in a single file. Lastly, batch command processing has been greatly expanded and is much easier to use than under CP/M.

- An entirely new concept compared to CP/M is the concept of ERROR HANDLING. This allows graceful recovery in situations where the user inadvertently made an error when issuing a command. A user can easily edit the offending command or select from several alternative actions, via a special program called an error handler. This intelligent technique of handling errors becomes even more important as the user begins taking advantage of the advanced command processing functions of ZCPR3.

- Greatly enhanced system security functions allow ZCPR3 to be used where public access to your computer system is necessary. Special features of the system can permit "dangerous" commands (such as erasing, copying, or renaming files) to be unavailable to ordinary users, yet when you or another privileged user is using the system these dangerous commands are available automatically. Areas of your computer can be restricted from access for storage of confidential and other sensitive information.

- Over 100 useful utility programs are available which make use of ZCPR3 extended functions. Many of these programs are equivalent to commercial programs for non-ZCPR3 computers which would cost the user hundreds of dollars to duplicate. Also, the utility programs generally share similar methods of usage, greatly easing the task of learning how to use these programs.

- The complete source code to ZCPR3, it's extended segments, and most of the utility programs is available. This permits the technically inclined user to fine-tune or extend his system in whatever manner he desires. No other command processor for any operating system has its complete source code (over two megabytes, in all) distributed.

1.2.2. Benefits of Z33 over Z30

While a conversion from Z30 to Z33 does not provide the spectacular improvement that the conversion to Z30 from CP/M did, it is still significant, especially for users making heavy use of advanced functions. We suggest that all users of Z30 make this conversion, as at a minimum you gain the benefit of the "bug fixes" present in Z33, whether or not you make use of its new features.

It should be noted that Z33 is a complete rewrite of Z30. Although functionally similar, Z33 is a completely new program and only bears a slight resemblance to Z30 in internal algorithms and code.

- Extended Command Processors and Error Handlers can be used simultaneously, which is not possible under Z30.

- A new type of transient program, call a type-3 program, can automatically load and execute at an address other than 100H, thereby preserving the contents of the TPA at address 100H for reinvocation with the GO command. Extended command processors, shells, and error handlers are good candidates for type-3 programs, as are "virtual resident" transient equivalents of CPR and RCP resident commands.

- The Z33 Command Processor reads the in-RAM Environment Descriptor to a far greater degree than the Z30 Command Processor does. Z33 determines environment parameters such as maximum disk and user, whether DU: forms can be accepted, and addresses of NDR, RCP, and FCP dynamically, which allows changing these parameters on-the-fly. This was not possible under Z30, which had these values hard-wired into the Command Processor at assembly time.

- The Z33 Command Processor examines each program loaded to determine whether it is a ZCPR3 utility. If the loaded program is a ZCPR3 utility, Z33 automatically installs it for your system. Z30 did not perform this automatic installation, with the consequence that uninstalled programs would not perform reliably.

- Speed of resolving commands can be significantly faster under Z33. Several options are available in this regard. The user can configure Z33 to not always search the current directory, which was not possible under Z30. Also, additional functions allow bypassing of the PATH for direct invocation of the extended command processor. The minimum path search algorithm is completely reliable and is now always turned on. These improvements result in a system that executes commands much more rapidly that Z30, especially in situations where the user is logged into a floppy drive and the path and/or extended command processors are located on hard disk or RAMdisk.

- Faster ZEX processing from shells such as VFILER and MENU/VMENU is available under Z33. Z30 is very inefficient and slow in this particular function.

- Secure configurations of Z33 are much more user-friendly than secure configurations of Z30. A Z33 secure configuration allows the non-privileged user to use allowable DU: forms, while rejecting non-allowable DU: forms. DU: checking is much more intelligent as items such as the presence or absence of a corresponding named directory entry and its password is significant as to whether a given DU: form is be accepted. Wheel-protected commands disappear when a non-wheel user is using the system, which allows extended command processors or error handlers to intercept attempts to use privileged commands. If the wheel byte is enabled, password checking of named directories can be bypassed.

- Several built-in commands enhanced and the SAVE command now accepts the number of sectors parameter described in the Z30 documentation. The RCP is scanned before CP-resident commands, which allows the user to override built-in commands with more powerful RCP-based ones. Several significant bugs fixed, overall reliability improved. The source code listing has been heavily annotated with comments, to benefit those seeking knowledge of Z33 internals.

1.3. The History of ZCPR

1.3.1. ZCPR1

"Don't you know about ZCPR!" We remember very well as a neophyte computer user being greeted with that exclamation by one of the experienced members of the computer club we had joined. He could not believe that someone would still be using CP/M. 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. Sometime around 1981 Richard Conn sparked the group's enthusiasm 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, and he took the lead role in the project. With the space opened up in the CCP, the group was 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 useless, 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. This was particularly a boon in the 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 and start all over again. With ZCPR, the REN and SAVE commands themselves would ask the user if the old file should be overwritten in such a case.

The original ZCPR was released to the public on SIG/M volume 54, dated February 2, 1982 (SIG/M, or Special Interest Group/Microcomputers, is the public-domain software distribution arm of the Amateur Computer Club of New Jersey). Several refinements were made to it 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.

1.3.2. 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 Valentine's Day, 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 maintain its integrity across 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 placed in another one of these buffers instead of hard-coding it into the command processor. In this way it 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! It included a concepts manual, an installation manual, a users guide, and a rationale manual (we guess Rick 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, we were using an Intel MDS-800 microprocessor development system, the computer for which Gary Kildall invented CP/M, and we remember very well bringing up this 8080 version of ZCPR2. It was marvelous!

1.3.3. ZCPR3

But ZCPR2 was by no means the end of the evolution. On Bastille day, July 14, 1984, only a year and a half after the original ZCPR, 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 one tenth of all the software ever released by SIG/M had come from one contributor -- Richard Conn!

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 automatic 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. If a whole, long

string of commands would have to be retyped because of a mistake in a single command, one would hesitate to take advantage of multiple command entries.

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.

Also diverging from previous tradition, ZCPR3 was the first ZCPR to be actively marketed as a commercial product. Richard Conn assigned the copyright to ZCPR3 to Echelon, Inc. in August 1984. Echelon has since developed several commercial products which incorporate ZCPR3, among which are Z-System (a complete disk operating system), and products which permit automatic installation of ZCPR3 on an existing CP/M 2.2 system (Z-Com). The combination of active marketing, support, and easy installation has made ZCPR3 the most popular ZCPR of them all, to date.

1.3.4. ZCPR31

The chain of refinements to ZCPR3 that have now led to the present version 3.3 started in March, 1985, when Jay Sage produced an experimental (unofficial) version of ZCPR3 called ZCPR31 for use on his Z-Node 13. It was modified so that the command processor would get the values for maximum drive and user from the environment descriptor.

The next and most significant advances occurred in August, 1985, when four further major changes were introduced:

1. The code was changed to prevent the infinite loop that resulted in Z30 when the loaded 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 ad infinitum.

2. 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.

3. 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. In this way, a ZCPR3 system could take advantage of both extended command processing and error handling.

4. So that one could easily change the program used for extended command processing to suit the work at hand, the code was changed to permit the name of the extended command processor to be stored in the message buffer. This is the only feature of ZCPR31 that has not been implemented in Z33. There are two reasons for this. First of all, the only space that was free in either the environment or message buffers was the so-called user-definable area at the end of the message buffer. This area was later used by the Z-Msg bulletin board program and TERM III telecommunications package, resulting in a conflict. Secondly, the introduction of the ARUNZ extended command

processor a month or so later rendered this feature largely unnecessary, since ARUNZ can effectively chain to any other extended command processor. At some time in the future, the message buffer should be extended to include, among other things, the name of the extended command processor (and perhaps the directory in which to find it) and the name and location for the submit command file (usually $$$.SUB).

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, introduced several new concepts, one was a new way to implement wheel-protected commands. 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 Hawley 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. 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 between Jay Sage and Bruce Morgen in July, 1986. They 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. Jay Sage then implement a small change in the code that made this problem vanish in a flash.

Two others contributed to ZCPR31. Steve Kitahata, sysop of Z-Node #57, put in a SAVS resident command that would save a specified number of sectors rather than pages. This code is no longer present in Z33, but its function has been incorporated into the SAVE command. Michael Bate performed some important housecleaning on the code, noting that many sections of code were being included when, because of the options selected, they were actually not needed. He added many conditional assembly instructions to keep code size down. This tradition has been followed in Z33.

1.3.5. ZCPR33

At the very end of January, 1987, Richard Conn decided to end his active participation with ZCPR3. Echelon approached Jay Sage about writing the official ZCPR version 3.3 release, based on the experimental ZCPR31. He 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 the remainder of this manual. The decision was made that Z33 would no longer make any attempt to support 8080/8085 computers. The code has been written using Zilog mnemonics, and extensive use has been 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. So far no use has been made of the index registers.

During these last three months of intense work, many individuals assisted by testing the code and offering suggestions. Besides those mentioned earlier, Roger Warren (sysop, Z-Node #9) and Dreas Nielsen have been particularly helpful.

One individual, Howard Goldstein, must be singled out for the transcendent role he has played in the development of ZCPR33. We have been in constant communication with him during the development process, and he has been a sounding board for all the new concepts. In his capacity as beta-tester extraordinaire, he has subjected the code to remarkable scrutiny, uncovering countless minor and occasional serious flaws. He has also suggested dozens of ways to shorten the code. Above all, working with him has been a pleasure and delight.

1.4. Definitions

Some important terms used in this guide may not be familiar to the reader--so we will define them here.

Command Processor - The command processor, or CPR, is the part of the operating system that obtains commands from the user and processes them. These commands may invoke an application program (word processor, spreadsheet, etc.) or perform a utility function (renaming files, showing directories of files, etc.)

Environment Descriptor - This is a special block of memory in the operating system which contains information about how the computer is configured. The ZCPR3 Command Processor and ZCPR3-specific programs refer to the environment descriptor to determine specific information about the system, information that is unavailable under ordinary CP/M. Several examples are: the number of disk drives, the CPU clock speed, and the number of rows and columns on the terminal screen.

Wheel - As part of system security, a specific memory location in the computer's main operating system is designated as the "wheel byte". The wheel byte is considered "off" when that memory location contains a binary zero value and "on" otherwise. Certain commands, especially ones, like ERA, that could damage the system, will not operate when the wheel byte is off.

This Page Left Blank

2. ZCPR33 COMMAND PROCESSING FUNCTIONS

This section of the user guide describes the enhanced command processing facilities provided by Z33 and explains how to make use of the new features.

2.1. Automatic Installation of Transient Programs

How many times have you picked up a new ZCPR3 program, tried to run it, and found that it would not work properly on your system? And how many times did you eventually discover that the trouble was that you forgot to install it using Z3INS? It certainly happened to us many times. Well, with Z33 it will not happen again. The command processor is now smart enough to install the program automatically just before it executes the code.

ZCPR3 programs need only one piece of information -- the address of the environment descriptor module in memory. The environment descriptor contains all the rest of the information a program needs to avail itself of Z-System services. ZCPR3 programs have this address imbedded in a standard location near the beginning of the code, shortly after a text string "Z3ENV" that identifies the program as a ZCPR3 program. What Z3INS does when it installs a program is to write the address of the environment descriptor into the file in the right place. The Z33 command processor knows the environment address and puts it into place in the loaded memory image.

Note that the installation is performed at run time, not at load time. Thus the GET command, for example, will load a file from disk without making any changes to the image in memory. It is only when the program is executed, either by a direct invocation or by the GO or JUMP commands, that the installation is carried out. Note also that the installation is performed no matter where in memory the file has been loaded. It is not limited to automatic installation at 100H.

To illustrate these points, consider the following unusual example. Suppose you create a single file (GROUP.OBJ) that is a concatenation of four individual 1K programs designed to run, let's say, at A000H, A400H, A800H, and AC00H. Each of these programs would have its own 'Z3ENV' header, and the entire collection would be loaded into memory using the command

```
GET A000 GROUP.OBJ
```

Then you could run the third program, for example, by issuing the command

```
JUMP A800 COMMAND-TAIL
```

The piece of code at A800H would be installed automatically as it was run.

One word of caution is in order in case you ever run programs from inside a debugger, such as DDT or DSD. After you have gotten used to the automatic installation feature of Z33, you will probably forget completely about the issue of installation. Unfortunately, the debuggers do not perform automatic installation, and the programs will not run correctly under the debugger

unless they have been installed for the system using Z3INS or ZRIP or have been manually installed from inside the debugger by poking in the correct environment address in the appropriate place. So you might not want to throw away ZRIP just yet. In fact, it is probably not a bad idea from time to time to run ZRIP on your entire disk to hard-install all the programs.

Z33 actually provides a second form of automatic installation that might be of use to programmers in some special situations. It involves passing the address of the environment module in a microprocessor register and is described in the programmer's reference manual.

2.2. Simultaneous Use of Error Handlers and ECPs

Z30 provided two very powerful and useful features to users -- error handlers and extended command processing. Unfortunately, it was an either-or situation; only one of these features could be used at a time. Now you can have both together!

Let's start with error handlers. As an option in Z30, when a command was entered that could not be found as a resident command in one of the ZCPR3 modules (the command processor, the resident command package, or the flow command package) or as a transient program (a COM file), a special error command could be executed instead. Error handler programs invoked this way could locate the bad command line and provide a variety of services to deal with the error. The simplest error handlers just reported the error, either cancelling any additional commands pending in the command buffer or skipping the bad one and going on with the others. The more advanced error handlers would allow the user to edit the command line so that the error could be corrected and command processing resumed.

Alternatively, one could configure the system with an extended command processor (ECP), whose standard default name was CMDRUN.COM. If a command such as

```
PROGNAME TOKEN1 TOKEN2 REST-OF-TAIL
```

could not be executed as entered because no file called PROGNAME.COM could be found, the command processor would effectively convert it to the command

```
CMDRUN PROGNAME TOKEN1 TOKEN2 REST-OF-TAIL
```

But what if CMDRUN could not process this command line correctly? Under Z30, all it could do was display an error message to the user and return control to the command processor. This meant that no error handling service could be made available. Even worse, any pending commands in the multiple command line buffer would proceed to be executed, even though their operation depended on the successful execution of the first command. This was not a good situation.

With Z33, a program designed to function as an ECP can determine that it was invoked as an extended command processor. Furthermore, it has a way to signal the command processor on return that an error occurred and that error handling services should be engaged. Thus the operation of the extended command processor is transparent to the user. The command

```
PROGNAME TAIL
```

(assuming there is no resident command of that name) will initiate one of the following actions. If PROGNAME.COM exists along the search path, it will be loaded and run. If not, the command processor will build the command line

```
CMDRUN PROGNAME TAIL
```

and, if CMDRUN.COM can be found and can execute this command line, it will do so. If not, either because the ECP could not be found or because the ECP could not process the command, the command processor error handling facilities will be engaged just as if there had been no extended command processor in the first place.

Two programs designed to function as extended command processors under Z33 are presently available: LX and ARUNZ. We will briefly describe the basic features of each. Both have additional capabilities that are not covered here.

2.2.1. LX Extended Command Processor

LX (Library eXecute) enables one to keep a number of individual transient programs (COM files) in a library called COMMAND.LBR. There are two reasons why one might want to do this.

First, programs take up space on a disk in fixed-size units, typically of lK, 2K, or 4K bytes. Even a tiny file takes up one full unit. When files are packed into libraries, however, the size of the storage unit is only 128 bytes. Consequently, libraries are often significantly smaller in size than he space that would be occupied by the individual files. Packing several dozen programs into a library file can save significant space on disk, up to hundreds of kilobytes.

The second advantage is that the entire library generally takes up only one or two directory entries, while the individual programs would take up one for each program. Since a disk has a fixed size directory, it can become full because all available directory entries are used up even though space for data storage is still available.

The syntax of LX when invoked manually is as follows:

```
LX PROGNAME TAIL
```

When this command line is executed, LX looks in COMMAND.LBR for the file PROGNAME.COM. If it finds it, it extracts the file from the library and loads it into the TPA for execution. It parses the TAIL just as the command processor would have and transfers execution to the loaded program. Thus, if LX.COM is renamed to CMDRUN.COM, one can execute a program that is in COMMAND.LBR, just as if it were available as an individual file on the disk,

with only a small speed penalty resulting from the extra step of invoking the extended command processor.

2.2.2. ARUNZ Extended Command Processor

ARUNZ (Alias RUN ZCPR) is an alias script processor. Standalone aliases are single COM files comprising two parts: the prototype command script and the script interpreter. The former is no more than 255 bytes long (and usually much less); the latter occupies the rest of the 1K file. The philosophy behind ARUNZ was to separate these functions. Rather than duplicate the script interpreter for each alias, a single program, ARUNZ.COM, contains a very powerful interpreter with much more extensive parameter evaluation capability. All of the prototype command scripts are kept in a single ASCII text file, ALIAS.CMD.

When the command line

```
ARUNZ NAME TAIL
```

is entered, ARUNZ scans the ALIAS.CMD file for a line beginning with NAME. If it finds it, the rest of that line in the file is taken as the alias script. Parameter expressions are expanded, any commands pending in the multiple command line buffer after the ARUNZ command are appended to the expanded script, and the resulting command line is placed into the command line buffer. Control is then returned to the command processor. If no line beginning with NAME is found, ARUNZ returns control to the command processor with flags set to signal the command processor to engage error handling. With ARUNZ.COM renamed to CMDRUN.COM, alias scripts in ALIAS.CMD can be executed just as if they were standalone aliases.

Besides automating many complex tasks using alias scripts, ARUNZ can be used to correct common typing errors and/or offer alternative names for commands. If you often type 'REMANE' when you mean 'RENAME', just enter a script

```
REMANE sys:rename $*
```

in ALIAS.CMD. Your mistyped command will then execute correctly. The disk space required for this alias is a mere 22 bytes, and the inclusion of the explicit directory where RENAME is kept (SYS in this example) will speed operation. If you don't like typing out the entire command 'CRUNCH', just enter the alias script (ARUNZ version 0.9C and later)

```
CR.UNC sys:crunch $*
```

Again, this costs just 22 bytes. It will allow you to enter the command as "CR", CRU", CRUN", or "CRUNC" in addition to "CRUNCH". If you also tend to mistype it, as I do, as "CRUCNH". just expand the script to

```
CR.UNC=CRUCNH sys:crunch $*
```

That only costs 7 bytes more. In general, the cost of ARUNZ aliases is so small that one can easily have hundreds of them.

2.2.3. Extended Command Processor Recommendations

We recommend using ARUNZ as the extended command processor in most circumstances. The main reason for this is that ARUNZ has enormously wide-ranging capabilities that can include the functions of other candidate ECPs such as LX or ZEX. For example, if there is a program called LXCMD in COMMAND.LBR, you can include the alias script

```
LXCMD LX LXCMD $*
```

in ALIAS.CMD. Then when you enter the command

```
LXCMD TAIL
```

and LXCMD.COM is not found, the following ECP command line will be executed:

```
CMDRUN LXCMD TAIL
```

With ARUNZ as CMDRUN, this will cause the alias script above to be executed, giving the command

```
LX LXCMD TAIL
```

just as if LX had been the extended command processor. At the same time, you might have a ZEX script file called ZEXCMD.ZEX that you would like to run when you simply enter the command name. This will happen if ALIAS.CMD contains the script

```
ZEXCMD ZEX ZEXCMD $*
```

Then when you enter the command

```
ZEXCMD TAIL
```

and ZEXCMD.COM is not found, the ARUNZ extended command processor will run the ZEXCMD alias, which will generate the command line

```
ZEX ZEXCMD TAIL
```

This is just what you wanted. The only disadvantage to this procedure is that you have to enter an alias script into ALIAS.CMD for each program in CONMAND.LBR and each ZEX script that you want to be invoked automatically in this way. When you add a new program to COMMAND.LBR, for example, you will have to add a new script to ALIAS.CMD also.

2.3. Enhanced Error Handling Under ZCPR33

The error handling facility in ZCPR3 is one of its most valuable and important features and one of the things that sets it apart so dramatically from either CP/M or MS-DOS. The Z33 command processor has augmented this facility dramatically both in power and in scope.

Under Z30, error handling was engaged only when a command could not be resolved as a resident command or located as a COM file along the search path. When that happened, the command processor, instead of executing the next

command in the multiple command buffer, would substitute an error handling command line stored in the message buffer (one of the ZCPR system segments). Although this error command line can contain a multiple command sequence, it typically contains a single command that invokes a special program designed to process the error that occurred. This error handling program uses a pointer stored in the message buffer to locate the offending command in the multiple command line buffer. The error handler program can then display that command line to the user and allow him to repair it by editing or substitution. No information is passed to the error handler, however, to indicate what kind of error occurred.

Error handling under Z33 has been greatly extended in scope. It is no longer invoked solely when a COM file cannot be found; many other situations cause it to be engaged.

First, several different kinds of errors can occur in the command processor itself. Failure to locate a specified COM file is just one of them. Other situations are:

1) the user attempted to change directories when directory changing was disabled for lack of wheel status;
2) an attempt was made to log into an improper directory, either because a directory with the specified name does not exist or because it specifies a user number or drive higher than what is allowed;
3) an incorrect password was entered at a prompt for password entry;
4) a command did not have a legal form (it either contained an ambiguous program name or an explicit file type);
5) an ECP called by the command processor was unable to process the command and returned control to the command processor;
6) a COM or other file could not be found in a situation that did not engage extended command processing (for example, when GET could not locate the file or an explicit directory was specified with a command);
7) an ambiguous file specification was given where one is not allowed (e.g., in the SAVE, GET, and REN resident commands);
8) a bad number was given with a resident command that expects a number (e.g., GET, JUMP, SAVE);
9) a file specified for a resident command (e.g., REN, TYPE, LIST) could not be found; and
10) a file attempted to load past the end of the available TPA.

In addition, error handling can now be invoked from outside the command processor. Any program code, including resident commands in an RCP or FCP and transient commands, can return control to the command processor with a flag set that tells the command processor to engage error handling.

Besides augmenting the SCOPE of error handling, Z33 has increased the POWER of error handling by making available information about the cause of the error. The ten error types listed above for command processor errors each return a specific code in a message buffer byte. External programs can also return error codes. Advanced error handlers will have the ability not only to display the bad command line to the user but also to give an indication of the possible nature of the error. The error codes presently defined and the interface specifications are given in the programmers' reference manual.

2.4. The Type-3 Environment

Under Z30 there were two types of program environments defined. Type-1 programs have external environments, the program header containing only the address of the memory-resident environment descriptor module. Type-2 programs have an internal environment, the entire environment module actually residing in the program. The latter was a vestige from ZCPR2 and is infrequently used. Since it can be useful as a way to allow a ZCPR3 program to work when run under standard CP/M, Z33 detects the type-2 environment and skips the autoinstall process (which would otherwise corrupt the internal environment descriptor).

For Z33 a new type-3 environment has been defined. In addition to the address of the environment module, it contains the address at which the program is designed to run. When Z33 loads a program, it examines the header. If it detects a type-3 environment, it reads the load address from the header and proceeds to load the program to the indicated address and execute it at that address. Here are some examples where programs with a type-3 environment are very useful.

Suppose you have just performed some program patching using a debugger and have just exited from the debugger. You want to save the new version of the program in a 36-page file named NEWPROG.COM. You enter the command

```
SABE 36 NEWPROG.COM
```

not noticing until it is too late that you typed the SAVE command incorrectly. The command processor, unable to find SABE.COM, loads and runs the extended command processor. When the ECP also is unable to process the command, it invokes the error handler. The error handler very nicely presents the erroneous line and allows you to edit the command line, replacing the 'B' with the intended 'V'. But what good would that do now? If the ECP program had not already wiped out the memory image you were about to save, the error handler would have done it! Well, with Z33 the ECPs and error handlers can have type-3 environments that allow them to load high in memory, at an address of 8000H for example. In that case, unless the memory image one planned to save was more than 32K bytes long (a rare situation), the image is still intact. The command line can be edited in the error handler and rerun. Thus extended command processors and error handlers are two prime candidates for type-3 environments.

Example two. You have just run your file copying program and copied some files. You want to rerun the program using the GO command to copy some more programs, but you can't remember the exact names of the other files. You run the DIR command to locate the files. Unfortunately, you forgot that the RCP with the DIR command was no longer loaded (perhaps you had loaded DEBUG.RCP to do some resident debugging work). Instead, the transient DIR.COM ran, and when you tried to repeat the copy operation using the GO command, it did not work; DIR.COM had overwritten it. With Z33 you can have transient versions of the resident commands with type-3 environments so that they operate high in memory and do not interfere with the lower TPA where typical transient programs operate. In this way the transient programs will function in many ways like resident commands. We call them virtual resident programs.

Some good virtual-resident candidates are: CD, DIR, ECHO (renamed from XECHO), ERA (renamed from ERASE), GOTO, REG, REN (renamed from RENAME), and WHL (renamed from WHEEL). A transient version of SAVE could also be quite useful. The ZSIG (Z System Special Interest Group) program FCP10 and the forthcoming Z33FCP flow control packages can be configured to load the transient IF processor (IF.COM) in high memory also, so that its operation will not affect low TPA (remember, the user may not always know when it -- and not the resident FCP command -- has run). Thus one might have an alias with a script including:

```
IF EX $1.COM
REN $1.BAK=$1.COM
FI
SAVE 36 $1.COM
```

The first line might be carried out by the resident FCP code or by IF.COM. With Z33FCP and an IF.COM assembled to run in high memory, it will not make any difference. Similarly, the REN command will be the resident version if one is available or a transient REN.COM otherwise. Again, with a type-3 environment, it will not matter.

Now for the last example. Some of you may be familiar with programs based on the technique pioneered by FINDERR. This is a program that runs immediately after another program and examines the memory image left behind by the first program. FINDERR was designed to look for the error count left behind by assemblers and compilers. Based on the error count, FINDERR would set or clear the program error flag in the message buffer so that flow control could be used to proceed differently depending on whether or not the previous operation was successful. A more recent program, MEX2Z, looks at the command line left in memory by the MEX communication program. MEX2Z can pick up any commands in the command line after the MEX exit command and can feed those into the ZCPR3 command line followed by another command of "MEX". This gives MEX a virtual shelling capability. To the user it feels as though he can enter operating system commands from inside MEX. Those commands are executed, and then control is returned to MEX. Amazing!

Programs of this type can face one insurmountable problem. Suppose a compiler or assembler keeps its error count at address 140H, or suppose MEX kept its command line buffer at 120H. There would be no way for a transient program, even if it were the minimum length of one record (80H bytes), to run without overwriting the very information it wants to examine. Again the type-3 environment comes to the rescue. By making the program load at a different memory address (say 200H in the above examples), the program will have no trouble examining the target memory. This application is quite real in the case of MEX2Z. One would really like MEX to return at the same baud rate that it was set to at the time the 'shelling' took place. The Z30 version of MEX2Z always returned to MEX at MEX's default initial baud rate. Since the baud rate byte is in the first record of memory, there was nothing MEX2Z could do about it under Z30.

2.5. Enhanced Command Search Control

The automatic path searching capability of ZCPR3 makes for a friendly, easy-to-use operating environment by freeing the user of the need to keep track of where programs are stored in the system. Under Z30, if the program you

requested could not be found in the currently logged directory, then the user-specified directory path would be searched.

While this path searching is very convenient, it does have one drawback -- it slows down system response. This is not so much a problem when the command really is in another directory along the search path. Then the search is worthwhile. But what if the command was mistyped? Then all the time spent searching along the path and attempting to process the command with the extended command processor is a waste. The system doesn't do anything useful for us until the error handler is finally loaded.

Another problem with path searching arises with the greatly increased use of alias scripts with the ARUNZ extended command processor. It is not unusual to find that more than half the commands on a system are ARUNZ aliases; sometimes the percentage is much larger. The time spent searching for these commands along the path before the ECP is invoked is wasted time.

Z33 offers solutions to these problems by giving the user greater control over when and how path searching is performed. Two improvements have been made in the basic operation of the path search.

2.5.1. Search of Current Directory (SCANCUR)

Under Z30, if the command processor was configured to allow directory specifications on commands, then the current directory was always prefixed to the symbolic path in the path buffer. Thus, when no explicit directory was included with the command, the current directory was always searched first. When an explicit directory was given, the current directory was searched second. For us and for many other users, however, the commands that we use most often are in directories other than the currently logged directory (on a RAMdisk, for example). Naturally, the system will respond faster (much, much faster in the case of a RAM disk) if this directory is searched first.

In Z33 the SCANCUR option works correctly. SCANCUR is defined in the Z33HDR.LIB file which is read during assembly. We strongly recommend setting SCANCUR to NO or FALSE so that the current directory is not automatically added to the path. If you want the current directory to be searched, you must include the expression '$$' in the symbolic path.

2.5.1.1. Precautions if using SCANCUR False

If SCANCUR is turned off, special precautions must be taken in the system initialization. These precautions are present in official Echelon implementations of ZCPR3, such as the SB180/DT42/AMPRO computers and Z-Com and "bootable disk" software products, so the following discussion only applies to users with unofficial implementations of ZCPR3 (such as those available on some remote access systems). If your coldboot code (special code in the BIOS that runs only when the computer is first turned on) does not set up a search path but instead leaves the path initialized to an empty state, then a startup command line such as "STARTUP" will not run, since the path will include no directory entries to search and the current directory is not automatically added to the path. There are two possible remedies in such a situation.

One remedy is to modify the coldboot code in the BIOS to set up an initial path of A0 by putting the single byte 01 at the very beginning of an otherwise empty path. The code for this would be

```
LD A,01 ; Put 01 in register A
LD (EXPATH),A ; Copy to first byte in path
```

Since this solution requires adding code to the BIOS, it may not be easy to implement, especially for anyone who is not an experienced assembly-language programmer.

The second solution is to change the startup command line. This can generally be accomplished with a simple patch; there is no need to rewrite the BIOS, with all of the complications that entails. Typical ZCPR3 BIOS coldboot routines set up the multiple command line with an initial command, usually an alias containing the rest of the commands required to initialize the system fully. The most common initial command line is "STARTUP". To make this run with no path defined, patch the BIOS code to change this command to %:START<0>" (the 'P' is replaced by a binary 0). You can use the techniques described in the chapter on installation to make this change using either a debugger program or a disk utility. If you need help doing this, your local Z-Node sysop is a good person to turn to. Obviously, you must also, then, rename the alias to START.COM. You must also make sure that the commands in this startup alias have some kind of directory prefix, at least until a path is set up. A typical START.COM alias might look like this:

```
:LDR SYS.ENV,SYS.RCP,SYS.FCP,SYS.NDR
:PATH RAM SYS RAM
<other commands>
```

The LDR command (with a colon prefix) must come before the path command (also with a colon prefix), because the system will not know where the path is until the SYS.ENV module has been loaded. The 'other commands' in the above alias do not need colon prefixes, since the path has been established by that point.

Even if you decide that you do want the current directory to be included in the search path, with SCANCUR off you can put it where you want in the sequence. We typically use a path that looks like:

```
RAM SYS ASM $$ RAM
```

The RAM disk is searched first, then the SYS directory on our system floppy, then the directory where we keep our assembly-language tools, and only then the current directory. Since the most frequently used commands are kept on the RAM disk, most commands are resolved immediately. The first three directories in the search path account for more than 95% of the commands entered. System speed improves dramatically.

2.5.1.2. Root of Path and Minimum Path

You may have noticed the odd looking repetition of the RAM directory in our sample path. The reason for adding RAM at the end is that the final path element, known as the 'root' of the path (no matter what its actual name is), is often used in a special way. For example, the command processor can be configured to look only in this root directory for the extended command

processor. Programs may be configured to look in the root directory for their auxiliary files.

If the RAM directory had already been searched unsuccessfully as the first path element, it would be foolish to search it again when the fifth element of the path was reached. (And if we are currently logged into the RAM directory, it would be searched three times!) Indeed, Z33 is smart enough not to do that. It has a feature called 'minpath' that eliminates duplicate entries before the search is actually performed. Z30 also offered this feature as an option, but it was coded incorrectly and generally could not be used. In Z33 minpath is always used.

2.5.1.3. Forcing a Search of the Current Directory

Now that we have improved efficiency for most of our commands, what about the case where we know the command we want IS in the current directory? Do we have to search the whole path before we get to it? No! By entering just a colon in front of the command, the current directory is automatically added to the beginning of the path. Since typing a colon can be a nuisance (it is a shifted character on most keyboards), Z33 offers the option (controlled by the ALTCOLON equate in Z33HDR.LIB) of typing another prefix character (default is period) instead. If the file is not located in the current directory, the rest of the path is still searched.

Note that the period (or other alternate character) cannot substitute for a colon elsewhere on the command line. In most cases, the alternate colon character will be recognized only if it is the first character of the command (i.e., the first character in the multiple command line or the first character after a semicolon command separator). If the ALTSPACE and ALTONLY options described below are both true, then the alternate colon character can be preceded by spaces.

2.5.2. Directory Prefixes

Now, suppose we go to the trouble to specify the exact directory where a file is located. Naturally we want that directory to be searched first. But consider further what would normally happen if we made a typing mistake and entered, for example

```
TURBO:TURVO
```

when we wanted to run TURBO pascal. In Z30, after TURVO.COM was not found in the TURBO directory, the rest of the path would be searched. Then the extended command processor would be invoked. Finally the error handler would come to our rescue. Z33 makes the assumption (controlled by the SKIPPATH equate in Z33RDR.LIB) that if we name a specific directory, we expect the file to be there and do not expect it to be somewhere else or to be an extended command process. Therefore, Z33 skips the path search and extended command processing. If the file cannot be found in the specified directory, then the command processor goes straight to the error handler for a quick fixup.

2.5.3. Direct invocation of Extended Command Processor

Finally, suppose we know that the command we are entering is intended for the extended command processor. As noted earlier, it is a waste of time to search

the entire path looking for the command. Even if it is found, it is not the one we want anyway! In Z33, commands entered with a special prefix are sent out for extended command processing immediately. There are several options for this prefix. The default choice is either a space character (since it is very easy to type) or a slash ('/'). If you prefer to be able to enter spaces freely on the command line, then the optional alternate character can be made mandatory (ALTONLY option). If you are satisfied with the space character only, you can save code in the command processor by disabling the alternate character (ALTSPACE option).

2.5.4. Summary of Path Search Rules

Here is a summary of the path searching rules for the standard Z33 configuration.

1. If a command is entered with no prefix and it is not a resident command, the normal path is searched. The current directory is included only if it is specified explicitly in the user's symbolic path. If the COM file is not found, the command is passed to the extended command processor. If that also fails, control is transferred to the error handler.

2. If a colon or a period prefix is placed before the command, everything is the same as described under (1) except that the current directory is added at the head of the path. Also, when any prefix of any kind is used with a command, resident commands are bypassed.

3. If a specific directory prefix is placed before the command, only that directory is searched for the command; if the COM file is not found there, then control is transferred directly to the error handler.

4. If the command is prefixed by a blank space or a slash, the command is sent directly to the extended command processor without any search for a COM file.

There is one set of exceptions to the above rules. Flow control commands, such as IF, ELSE, or FI, are ALWAYS processed by the FCP module, no matter what kind of prefix is placed in front of them.

2.6. Command Acquisition Hierarchy

ZCPR3 offers a significantly wide range of options regarding how commands can be generated by the system. An obvious function of ZCPR3 as a command processor is to seek the next command to process. This search for new commands is called command acquisition, and there is a specific priority system (hierarchy) which determines what the next command to be processed will be, as there may be different commands waiting to be executed on several levels. It is worth noting that ZCPR3 possesses more levels of command hierarchy than any other command processor for personal computers. To fully use some of ZCPR3's advanced functions, you should understand this hierarchy.

2.6.1. Z30 Command Acquisition Hierarchy

Under Z30, each time the command processor was ready for another command, it would seek it from the following sources in the following order:

1. the multiple command line buffer. If any commands remained in the multiple command line buffer, they would be executed first.

2. a submit file. Once the multiple command line buffer was empty, if the submit facility was enabled, the command processor would search for a file called '$$$.SUB' on the A drive and current user area. If one was found, the last record in that file would be read and used as the command line. This record would then be deleted from the file.

3. the shell stack. The shell stack can contain several (usually four) command lines in a push-down stack. If the stack was not empty, the command line on the top of the stack would be copied to the command line buffer and executed.

4. the user. Finally, if none of the above sources could provide a command line, as a last resort the user would be prompted to enter a command!

One problem with the above scheme was the way it handled the operation of ZEX, the memory-resident batch command processor (similar to submit). If no shell command and no submit file were present, everything was fine. ZEX would provide the input to the user prompt. A problem arose, however, when a shell was running. Unless special action was taken, the shell command would run and would take its input from a running ZEX script. This would usually not produce the intended results. Consequently, shells like VFILER and MENU included special code to detect the presence of a running ZEX job and to pass the next ZEX input line to the command processor. This meant that the shells would be reloaded from disk after each ZEX command line, only to discover that ZEX was running. They would then pass on the command line and return control to the command processor. This pointless loading of the shell program resulted in very slow and annoying operation, especially on floppy-disk-based systems.

2.6.2. Z33 Command Acquisition Hierarchy

Under Z33, a different philosophical viewpoint has been taken. It may not satisfy all users, but we think that it will satisfy many more than the old scheme did. Z33 treats an entire ZEX script as if it were a single virtual (possibly very long) multiple command line. Once a ZEX job is started, it maintains control and continues to feed its input to the command processor until the job is completed. The command source hierarchy under Z33 is, thus, as follows:

1. the multiple command line buffer
2. a ZEX job
3. a submit file
4. a shell command
5. user input

A command from any level in the hierarchy can immediately initiate a process that is higher in the hierarchy. A command in the multiple command line buffer can spawn additional commands (this is what aliases do). ZEX jobs can supply a series of multiple command lines (but not, at present, start another ZEX process). A submit job can supply command lines or initiate a ZEX process or invoke another submit job (the second submit job appends its commands to those of the first). A shell can generate command lines, start a ZEX process, initiate a submit job, or load another shell onto the stack and begin to run that shell instead. Finally, the user can start anything from the command prompt.

Conversely, commands from a given level may produce strange (though predictable) results when they initiate processes at a lower level. If a ZEX job issues a submit command, the '$$$.SUB' file will be created, but it will not take effect until the entire ZEX script has finished executing. If a ZEX job invokes a shell program, the name of the shell should be installed on the shell stack, but the shell should not begin to operate until the ZEX script has run to completion. (With recently written or updated shells, however, we have discovered that this does not generally work. A very simple addition to the shell coding is required to make them work predictably in this situation.) Many users have been puzzled by what happens when a shell command is included in the middle of a multiple command sequence. According to the above hierarchy (shells are below the multiple command line), the shell will install itself on the shell stack but will not begin to run until the rest of the commands in the existing command line have been completed.

2.6.3. Improvements to SUBMIT Processing

Since we have been discussing a change in the way ZEX batch processing operates, this is a logical place to mention some very important changes in the way submit processing is handled.

Under Z30, SUB.COM created the $$$.SUB files of commands on drive A in the current user area. This caused some serious difficulties. If a command in the submit job changed the logged-in user number, the command processor would lose track of the $$$.SUB file, and batch processing would stop. Even more bizarre behavior would occur if one later returned to the former user area and initiated a warm boot: the batch job would start up again!

Z33 has fixed these problems by writing and looking for the $$$.SUB file in user 0. Submit jobs can now freely change user numbers just as ZEX scripts could.

Submit control flags, paralleling those for ZEX, have been added to the message buffer. It is now possible for any program to detect that a submit job is in process. An input redirection flag, like the one for ZEX, could be used by an advanced version of XSUB (not yet available) to temporarily halt input from a submit/xsub job.

In the past we chose to omit the submit facility from our ZCPR3 command processor implementation. With these changes in Z33, the submit facility has become so much more attractive that we now do include it (its major advantage over ZEX is that the full TPA is available).

There is now an option to control the display of submit prompts on the screen. If the SUBNOISE equate is set to 0, no command line input will be echoed, and the entire submit script will run as if it were a single, long multiple command line. If SUBNOISE is set to 1, then echoing of the command lines is controlled by the QUIET flag in the environment descriptor. If the QUIET flag is set, command lines will not be echoed. Finally, if SUBNOISE is set to a value higher than 1, submit command lines will always be echoed, as they were in Z30.

2.7. Command Resolution Hierarchy

In the previous section we discussed the hierarchy of sources from which the command processor seeks new command line input. Here we will consider the hierarchy by which the command processor attempts to execute each individual command after the command has been acquired. Again, this is a topic which you should understand to fully use the advanced features of ZCPR3.

2.7.1. System Command Resources

In a ZCPR3 system, there are two general classes of commands: resident commands and transient commands. Resident command code is found in memory, and transient command code is found on disk.
There are three sources of resident commands: the command processor itself, the resident command package (RCP), and the flow command package (FCP). Implementation of the RCP and FCP are each optional, and, when implemented, the code in the module can be changed at any time by loading new packages into memory from disk. The command processor code is fixed.
Transient commands are either direct commands (COM files) or extended commands processed by an extended command processor. The way the command processor code is currently written, the extended command processor code must be contained in a COM file; it cannot come from from one of the resident command modules. One might want to change this in the future so that one can have a fast resident extended command processor.
A command with a particular name could be found in any or all of these sources. When presented with a command, the command processor tries to resolve it according to the following hierarchy:

1. First the FCP is scanned to see if the command is a flow control command. The command verb (the name without any directory or other prefix) is presented to the FCP no matter what prefix might be attached to it. It makes no difference if there is a colon or a period, a space. or a DU or DIR (named directory) prefix. The FCP always gets the first shot at any command. There are two related reasons for this. First, flow control commands can change the flow state, and the flow state determines whether other commands are executed or flushed. Secondly, flow control commands must be executed even if the current flow state is false, since they are the only commands that can change a false flow state back to a true flow state.

2. If the current flow state is false, the current command is flushed, and the next command is processed.

3. If the command verb is not found in the FCP command table, and the command has no prefix (colon, dot, space, slash, or explicit directory specifier) then the other two resident command modules (RCP and command processor) are scanned. If the command had a prefix, the resident modules other than the FCP are skipped. In Z30 the command processor was scanned before the RCP, but this order has been reversed in Z33. This change was made so that one would have the option of loading an RCP module with a more powerful version of a command processor-resident command and have the RCP command effectively replace the command processor one. Before, once a command was implemented in the command processor, one was stuck with it for good.

4. If the command could not be found in any of the resident modules or if the command verb had a prefix, the command is treated as a possible transient program. If the command has a space or slash character as a prefix, then the command is sent immediately to the extended command processor without any attempt being made to resolve it as a normal transient program. There is also one other special case. If the command verb is a null command (a prefix only was present), then the command is interpreted as a command to change the logged-in directory. Otherwise, the program is searched for as a transient program according to the rules described above under the section on paths.

5. If a COM file with the name of the command cannot be found anywhere along the search path, and if the command did not have an explicit directory prefix, the entire command line is passed on to the extended command processor (ECP). The ECP is itself a transient program and must be located by searching the disk directories. The path used for the ECP search can either be the entire path or just the root element. This is controlled by the ROOTONLY equate in Z33HDR.LIB. Note that the root element is not necessarily a directory with the name ROOT; it is simply the very last element specified in the symbolic path. The ROOTONLY option is generally preferable, since it will result in faster system response.

6. If the ECP cannot be found, or if the ECP is unable to perform the command either (and has Z33-compatible facilities for returning to the command processor), error processing is carried out using an external error handler, if one is installed, or internal error handling code in the command processor. The details of error processing are described elsewhere.

2.7.2. Flow State and Shell Stack-Generated Commands

There is one important exception to the rules described above. When the specified command is running as a command processor-invoked shell command (not one entered on the command line), special rules apply. The need for special rules can be appreciated from the following example. Suppose that at the end of a command line entered from a shell, the flow state is left in a false condition. Then when the processor attempts to run the shell command from the top of the shell stack, the shell command will be flushed. The command line will again be empty, and the shell command will again be copied to the command line. Again it would be flushed, and so on forever.

Under Z33, there are two alternatives controlled by the SHELLIF equate in Z33HDR.LIB. Setting this equate false gives the normally recommended behavior as follows. First, if the current command is running as a shell and is itself a flow command (this can happen if the shell command line has multiple commands or contains an alias command), then it is simply flushed. Shell command lines are not allowed to contain flow control commands (actually, it can contain them, but they will not be processed). Second, if the command is not a flow command, then it is run no matter what the current flow state is. Thus the shell will run even if the current flow state is false. With SHELLIF off, each command in a multiple-command shell will think it has been invoked as a shell command. This makes it possible for the shell to be an extended command process, such as when the shell is in COMMAND.LBR or is an ARUNZ alias.

If SHELLIF is enabled (not recommended), then flow commands in a shell command sequence ARE processed. In order for this to be done and make sense, two other things have to be done. Since flow commands are recognized in the shell sequence, entering in a false flow state would cause a problem. Consequently, with SHELLIF on, the flow state is reinitialized each time a shell starts to run. Thus all user command sequences with flow control must be completed on a single command line (or virtual command line produced by ZEX or submit). Since the flow state is reset for each command that thinks it was invoked as a shell, multiple-command shell commands must be handled in a way that makes only the first command think it is a shell. The other commands must think that they are ordinary, user-invoked commands. Quite bizarre behavior can result from this kind of processing; that is why we do not recommend using this option. Some users have a special application for it, however.

2.8. Directory References and System Security

In this discussion we will use the term "directory" in the UNIX sense to refer to a logical group of files. In the case of ZCPR3 this means a specific user area (or user number) on a specific disk device.

Z33 has made extensive and significant changes in the way directories can be referenced in commands and how security (if enabled) is handled. First we will present some general background information and describe the situation as it existed in Z30.

ZCPR3 supports two basic forms of directory reference, the disk/user or DU form and the named directory or DIR form. We will assume that the reader is already somewhat familiar with user areas and disk identifiers. The DU form is native to CP/M, which knows about disk drives from 'A' to 'P' and user numbers from 0 to 31 (though there are restrictions on user numbers above 15). The drive part of the reference is a physical reference. Although the association between letter names and physical devices can, in some systems, be modified in software, all user numbers associated with a given drive are on the same physical device.

Named directories, on the other hand, are purely logical constructs. The named directory register (NDR) module in a ZCPR3 system, which is loaded by a user command, contains a mapping of directory names to drive/user values. The user can load different sets of directory associations at different times. When the DIR form is used, the command processor looks for the name in the NDR and substitutes the drive and user values. Only drive/user values are used in the

actual file references, that is, in the file control blocks constructed by the command processor.

Named directories provide two different and important functions, one is convenience. It is much easier to remember that one's assembly-language tools are in ASM and one's wordprocessing files in TEXT than it is to remember that the directories are A1 and B31 You might have a floppy disk containing letters to three friends: Bill, Bob, and Bud. You may have trouble remembering which ones are in which of the user areas 1, 2, and 3, but you will not forget which are in the named directories BILL, BOB, and BUD. Now that there are some very powerful tools (e.g., LOADNDR) for automatically setting up directory names in an adaptive way, named directories are quite useful on floppy as well as hard-disk systems.

The second purpose of named directories is to provide security. When Z30 is assembled, it is configured for maximum drive and user values. References to directories outside that range are not accepted (they are interpreted as references to the currently logged directory). This feature could not be used conveniently to provide security, however, because it offered no flexibility. The maximum drive and user values were hard-coded into the command processor and could not be changed depending on the privilege level of the user.

2.8.1. Named Directory Passwords

Named directories offer a more flexible means of controlling access to areas on a system. The user can access a named directory even if it refers to a drive/user area that is beyond the normal bounds. Each directory name can have an optional password associated with it. Whenever the command processor parses a file specification with a passworded named directory, it will ask the user for the password. If the correct password is given, then the corresponding drive and user values will be used for the file reference. If the password is entered incorrectly, then in Z30 the current drive and user will be used, effectively denying access to the protected directory. In Z33 the error handler will be invoked so that the user can try again (in case he mistyped the password) or abort the command or command line. Since the contents of the named directory registers can be reloaded, very flexible security can be achieved using named directories.

Note that named directories and password checking do have their limits. The command processor normally parses only three tokens in a command: the command itself and the first two tokens in the command tail. A directory reference with the command name is used to determine in which directory to look for the command. Directory references for the first two command-tail tokens determine what drive and user values will be stored in the two default file control blocks that the command processor builds for use by the program. If there are additional directory references in other tokens, they will be handled only by the program itself and, thus, may or may not be interpreted correctly as named directories or have their passwords checked. Most ZCPR3-compatible programs, but not all, which use additional command-line tokens do take care of password security.

The command processor has many options concerning directory forms and security. It can be configured to accept both DU and DIR forms, either one alone, or neither. If it accepts both, it can attempt to recognize a directory reference first as a named directory or first as a DU form. Here is

an example of where the order makes a difference. Suppose you have a named directory called 'C' associated with drive B, user 3 (perhaps you keep your C language files there). If you are logged into A1 and use a file specification of "C:PROG", it will refer to B3 if the DIR form is interpreted first or C1 if the DU form is interpreted first.

2.8.2. Z33 Security Improvements Philosophy

Many improvements in Z33 have been introduced to deal with security issues. However, this was not done, as one might think at first, to provide increased security. Rather it was done to make it possible to have a secure system (a remote access system for example) while maintaining the user-friendly convenience associated with nonsecure ZCPR3 systems. We have always been troubled by secure ZCPR3 systems because they give the impression that ZCPR3 makes a system difficult or complex to use, when, in fact, the very opposite can, and should, be the case.
Here are some of the problems that used to exist with secure systems and the way the problems have been solved in Z33.

2.8.3. DU References to Directories with Names

Named directories can be a pleasant optional convenience, but they quickly become a burden when their use is made mandatory. The fact is that in many (even most) cases, DU references are quicker and easier to type. Unfortunately, because of Z30's fixed maximum drive and user limits, secure implementations under Z30 were generally forced to disallow the DU form.

In Z33 this problem is dealt with in several ways. First of all, the command processor can use for its maximum drive and user limits the values specified in the system environment descriptor. It was ironic with Z30 that all the utility programs used the environment module to get information about the system, but the command processor had all the information hard-coded in (this topic is dealt with more generally in the next section). With a Z33 secure system, for example, the maximum user number can be set to 5 for normal users, 7 for certain privileged users, and 31 for the sysop. The limits are changed simply by loading a new ENV file or by poking new values into the ENV module in memory.

Even with the ability to change the maximum drive/user values, there are cases where stricter limits are required. Suppose, for example, that you want users to have access only to the areas B1 and C1. Clearly this cannot be done with drive and user limits only. The drive limit would allow drive A to be accessed, and the user limit would allow access to user 0. Whenever Z33 is assembled with the code for recognizing the DU form, even if it is temporarily disabled (more on that shortly), a DU expression that refers to a directory that COULD be accessed using a DIR form is accepted. Thus, if in the above example, B1 and C1 have the names LOWLY and GUEST, with no passwords, then the expressions B1: and C1: will be accepted. The expressions BO: and A1:, however, will not be accepted. If LOWLY and GUEST do have passwords but 1) the wheel byte is on and 2) the WPASS option has been selected to bypass password checking when the wheel byte is on, then the forms B1: and C1: will be accepted despite the passwords.

How can the code for DU references be present but temporarily disabled? Z33 can be configured to respond to the DUOK flag in the environment descriptor. If this flag is turned on, normal parsing of DU expressions is performed. Drives and users up to the maximum values, either specified in the environment or hard-coded into the command processor, will be accepted in addition to those allowed because of corresponding named directories. When the DUOK flag is turned off, only those directories allowed because there is a corresponding named directory will be accepted. This is ideal for maximum flexibility in a secure system.

2.8.4. Unconditional Acceptance of Current Directory

There are several other annoying things that can happen in a secure system. Suppose you have logged into directory PRIVATE (A3) using the password SECRET. Consider an ARUNZ alias with the following script that allows the non-ZCPR3 program CPMPROG (renamed to CPMPROG0) to accept a named directory reference. We assume that CPMPROG, like many modern CP/M programs, has been extended to recognize the DU form. Here is the script:

```
CPMPROG cpmprog0 $d1$u1:$:1.$.1
```

The complicated parameters in this example extract the drive ($d1), user ($u1), file name ($:1), and file type ($.1) from the first token. With this alias, when you enter the command

```
CPMPROG GUEST:NAME.TYP
```

ARUNZ turns it into the command line

```
CPMPROG0 C1:NAME.TYP
```

This works very nicely. But now suppose you enter the command

```
CPMPROG NAME.TYP
```

to refer to a file in the current directory. This turns into the command line

```
CPMPROG0 A3:NAME.TYP
```

Unfortunately A3: is not an acceptable reference (it refers to a named directory beyond the allowed ranges and with an active password). Note that this directory reference (A3:) is to the directory that the user is currently logged into. In this situation, Z33 has been enhanced to not prompt for passwords since it can be assumed that being currently logged in to a given passworded directory indicates the user should be allowed to remain there and refer to it without further inconvenience. This is a significant improvement over Z30. Hence this basic rule: In Z33 all references to the current directory, whether using the DU or the DIR form, will always be accepted, without restriction. If the reference uses the DIR form, the password will not be requested again. Once you leave this directory, of course, full security will be exercised if you attempt to return. Also, if the code for processing DU forms is entirely omitted from the command processor by setting the ACCPTDU option to false, then no DU references will be accepted under any conditions.

2.8.5. No Password Checking Under False If Conditions

Consider another alias that could give trouble in a secure system. This more complex script prompts the user as to which of two protected directories he wants to enter:

```
script ENTER:

     IF INPUT ENTER PRIVATE DIRECTORY 1 (Y) OR 2 (N) ?
        CD PRIVATE1:
     ELSE
        CD PRIVATE2:
     FI
```

Under Z30, no matter how one answered the prompt from "if input", passwords for both private directories would be requested. Even though the flow state is false during the execution of one of the two 'CD' command lines, the line must be parsed in case it turns out to be another flow control command. This example is simple compared to some real-life experiences with Z30, which when the user referred to a password-protected directory, would ask for the password six times, even though only one command was actually going to be executed. In Z33, the command processor is smart enough to skip checking passwords when the current flow state is false. To make sure that the user cannot finesse information from a private directory during false IF states, the command processor does not just bypass password checking; it also substitutes the current directory for the specified one. This is designed to take care of the situation in which commands like the following are entered:

```
     IF FALSE  ; Set a false IF state to bypass
               ; ..password checking
     OR EXIST PRIVATE:FN.FT Now see if the file exists in the
               ..private area while we will not
               ..be asked for the password
     IFQ       Check resulting IF state
     FI        End the process
```

The command processor will treat this command sequence as if it had been:

```
     IF FALSE
     OR EXIST FN.FT
     IFQ
     FI
```

Thus an OR command may not work when it refers to a password-protected directory, and one should avoid ever using the OR command with such references. Put still another way, the following sequence is not reliable:

```
     IF EXIST PRIVATE1:FN.FT
     OR EXIST PRIVATE2:FN.FT
```

For the first test, one will be asked for the password. If that test is false, then one will not be asked for the password to the PRIVATE2 directory, and the current directory will be searched for FN.FT. This problem can be avoided in most cases by using the De Morgan boolean equivalent command pair based on AND instead of OR:

```
IF ~EXIST PRIVATE1:FN.FT
AND ~EXIST PRIVATE2:FN.FT
ELSE
```

Here, if the first test is false, the outcome of the second test is irrelevant anyway, so it is just as well that the command processor not pester us for the password to directory PRIVATE2.

2.8.6. Special Options for Directory References

There are several special options in Z33 for handling directory references. We mentioned WPASS above. This option allows all password checking to be suspended when the wheel byte is on. With the WDU option enabled, changing directories using commands like "DU:<cr>" or "DIR:<cr>" will not be allowed when the wheel byte is off. With WPREFIX enabled, explicit directory prefixes on commands (e.g., "B2:PROGNAME" or "MYDIR:PROGNAME" or even ":PROGNAME") will be ignored when the wheel byte is off; the commands will be searched for only along the path, just as if no prefix had been present. This option can be used to prevent a user from running a COM file that happened to find its way into a directory where it did not belong. Finally, the PWNOECHO option allows one to prevent the password from appearing on the screen as it is entered. This is done by temporarily patching the BIOS console output routine (the jump opcode is changed to a return). When the password entry has been completed (and also at each warmboot, just in case anything went wrong along the way), the BIOS is patched back to normal.

2.8.7. Passing Illegal Directory References to an ECP

There is a special option (BADDUECP) that determines how the command processor will deal with attempts to log into a non-existent or off-limits directory. If BADDUECP is off, the command will be treated as an error and sent immediately to the error handler. If BADDUECP is on, then the command is sent instead to the extended command processor. If ARUNZ is the ECP, the following alias script is very handy for dealing with this situation. The alias name is

```
?:=??:=???:=????:=?????:=??????:=???????:=????????:
```

This alias is placed at the end of ALIAS.CMD. It recognizes any command with from 1 to 8 characters followed by a colon (i.e., any DU: or DIR: command). The command script associated with this alias name is

```
ECHO d%>irectory %<$0%> is not a valid directory. %<t%>he following
ECHO %>directories are available:
ECHO
PWD
```

If the user enters a command such as "BADDIR:<cr>", then the following message is displayed in front of the PWD listing of the directories on the system.

```
Directory BADDIR: is not a valid directory. The following
directories are available:
```

On a remote access system, this is more useful than having the error handler invoked, since it gives the user the information he needs to enter a correct command.

2.9. Command Processor Response to the Environment Descriptor

When ZCPR3 was introduced, one of its most striking concepts was the use of the environment descriptor to provide a unified description of the system to all of its components. How strange, then, that the command processor, the heart of the system, did not use the environment descriptor for its configuration information! With Z33 this has been changed to a significant extent.

It is probably not feasible, because of the additional code it would require, for the command processor to get all of the information it needs from the environment descriptor. In the CPR there are numerous references to the multiple command line buffer, message buffer, path, external file control block, external stack, wheel byte, shell stack, named directory register, flow control package, and resident command package. Computing the required addresses each time they are needed would be far too cumbersome.

Z33 has tried to reach a compromise solution, making the command processor responsive to some but not all information in the environment descriptor. This information is of two types -- system characteristics and module addresses.

2.9.1. Equates Controlling Command Processor ENV Access

There are several system characteristics defined in the environment descriptor that the command processor should be able to respond to. Because extra code is required to implement these features and not all users will require them, they are all optional and controlled by equates in the Z33HDR.LIB file. The DUENV (DU from ENV) option allows the command processor to determine from the environment the highest drive and user number to be recognized. The ADUENV (Allow-DU from ENV) option makes the command processor's acceptance of the DU form follow the status of the DUOK flag in the environment. If this flag is set, the DU form will be recognized; if it is not set, then only named directory references will be accepted. Since even with DU processing disabled by the DUOK flag, the command processor can still recognize DU expressions if they refer to directories with names, one may or may not want the prompt to show the DU value when DUOK is off. The INCLENV (INCLude in prompt from ENV) option makes the display of the drive/user in the command prompt dependent on the DUOK flag also.

It is impractical to have the command processor get the addresses for ALL system modules from the environment descriptor. We have assumed a 'min' system configuration with a multiple command line buffer, message buffer, shell stack, path, external stack, and external file control block in addition to the environment descriptor itself. A wheel byte is assumed if any option requiring a wheel byte is enabled. These system modules must be located at addresses fixed at the time the command processor is assembled. Other than the decision to include or omit these modules, no other options were generally considered under Z30. Although for some, like the shell stack, the size could be specified, the standard configuration was almost always used.

2.9.2. Dynamic Sizing of FCP/RCP/NDR

With the Z33 command processor, the locations of the three largest system modules -- the RCP (resident command package), FCP (flow command package), and NDR (named directory register) -- can be determined from the environment. If the system is configured with these three modules contiguously located, then the total buffer space they occupy can be traded off on a dynamic basis. There might, for example, be a total of 24 records or 3K bytes allocated for the three modules. Under one set of circumstances, the RCP might get 17, the FCP 5, and the NDR 2 sectors. This allows one to have extensive RCP and FCP commands but only 14 directory names. At another time, one might want to change the balance to 16, 4, and 4 sectors for the three modules, respectively. Now there could be 28 directory names but fewer RCP and FCP commands. By setting the address of a module to zero in the ENV, the corresponding feature in the command processor can be disabled.

If you decide to make use of this capability, great care must be exercised, since the command processor accesses these modules for each command it processes. Consequently, the changes cannot be made piecemeal. If you were to load only a new environment descriptor, and it contained a different address for the flow command package, the system might well crash when it tries to interpret whatever code is at this new address as an FCP module. It is generally best to make all the changes in a single command, such as the following:

```
LDR NEW.ENV,NEW.NDR,NEW.FCP,NEW.RCP
```

This command loads the new environment first. Then, with that established, it can proceed to load all the other new modules. It is safest to put commands like this into aliases so that no mistakes can occur (such as mistyping the name of one of the modules in the above command). You might have a general-purpose alias called SETENV that would be invoked with a number to select the environment configuration. Here are some examples.

```
SETENV 1
SETENV 2
```

Part of the script for that alias would be the following:

```
IF EXIST SYS$1.ENV
   LDR SYS$1.ENV,SYS$1.NDR,SYS$1.FCP,SYS$1.RCP
   ECHO ENV $1 ESTABLISHED
ELSE
   ECHO ENV $1 NOT FOUND
FI
```

A few extra commands would be required to make sure that it referred to the correct directory areas for the files.

3. COMMAND-PROCESSOR-RESIDENT COMMANDS

In Z33 the resident commands are considered to be secondary elements to be included only to the extent that space remains after the essential command processing features have been implemented. Command processing functions can be performed only by the command processor; the utility functions can be performed in other ways. In the forthcoming companion resident command package. Z33RCP, all of these commands except for GET, GO, and JUMP will be available.

3.1. Overview

In all there are ten resident commands in the command processor: DIR, ERA, GET, GO, JUMP, LIST, NOTE, REN, SAVE, and TYPE. They can be divided into two classes according to the extent to which they make use of command processing code to carry out their functions.

Three of the resident commands -- GET, GO, and JUMP -- are intimately tied into the command processor code, and they should be the first commands chosen for inclusion if their functions are desired.

Implementing GET in an RCP would involve extensive duplicate coding, since GET utilizes all of the command processor code related to the execution of transient programs except for the final call to the loaded code. This includes determination of the load address, building the search path, searching for the file, and loading the file to the specified address.

GO and JUMP, on the other hand, make use of the one part of the command processor code that GET does not use -- the code for executing a loaded program. This includes automatic installation of the loaded code. The JUMP command also makes critical use of the command processor's parser to reparse the command tail omitting the load address. With the new defined entry point into the command processor's parsing code, it is now possible to implement the JUMP command in the RCP without a great penalty in code. Nevertheless, GET, GO, and JUMP together require only a little more than 60 bytes of additional code plus 18 bytes for their entries in the command dispatch table, so it does not cost very much to include them.

The remaining commands -- DIR, ERA, LIST, NOTE, REN, SAVE, and TYPE -- are largely independent program functions stuck into the command processor. Formerly, including them in the command processor offered two advantages: first, they responded faster because they did not have to be loaded, and, second, they did not use memory in the TPA. The latter factor was convenient because it made it possible to rerun the previously loaded transient command using GO even after using one of the resident functions had been run. Not using the TPA was critical for the SAVE command, whose function of copying the memory image in the TPA to a file could not be performed by a program which, itself, had to be loaded into the TPA.

All of these commands can be performed equally well, however, as resident commands in an RCP, where the greater amount of available memory can be used to give the commands more power and flexibility. The new type-3 environment, furthermore, makes it possible to have transient versions which, though taking time to load, will not interfere with low TPA memory. One can even have a SAVE.COM now. Consequently, these functions should be included in the command processor only if space remains after all desired command processing options have been implemented.

We will now discuss the changes that have been made in the way these resident commands function under Z33. For overall descriptions of the commands, the reader should consult Richard Conn's ZCPR3: The Manual. The NOTE command is not discussed below, since it has not changed at all.

3.2. DIR

The DIR command always supplied an automatic file specification of "*.*" when the command was used without any tail. However, if the user wanted to include files with the SYS attribute or to see only those files, the commands had to be entered as "DIR *.* A" or "DIR *.* S". If the SLASHFL option is enabled in the Z33HDR.LIB file, then the following commands will accomplish the same functions: "DIR /A" and "DIR /S".
There is now an optional wheel restriction on the display of files with the SYS attribute. If WHLDIR is enabled, the 'A' and 'S' options will be ignored when the wheel byte is off.

3.3. ERA

The file erasing command is essentially identical to the one in Z30. There is, however, greater flexibility in the choice of the character that invokes the inspection/verification option. Z30 used 'IV', but the transients (e.g., ERASE, RENAME, MCOPY) and the RCP version of ERA all use 'I'. With the default setting of INSPCH to ' ' (space character) in the Z33HDR.LIB file, any character at all ('I' or 'V' or anything else) in the second command line token will enable prompting. If you prefer a specific character, change the setting of INSPCH to the uppercase character or symbol of your choice.
A second change is that ZEX input redirection is turned off during the operation of all resident commands so that the inspection/verification prompt will not be answered by characters in the ZEX script.

3.4. GET

The GET command loads a file or transient program in exactly the same way that it would be for execution except that it is loaded to the address specified by the user, no matter what address it would be loaded to otherwise. The syntax is:

```
GET HEXADDR UNAMBIGFN
```

Here are some examples:

```
GET 100 UTILITY.COM
GET 1A35 FIX:PATCH.OVR
GET A00 3:SYS.RCP
GET 180 :LOCAL.FIL
```

Note that ZCPR3: The Manual contained an error in the description of the GET command. The address is the full hexadecimal address and NOT the page address. Do not use an 'H' after the address. Files can be loaded to any address, not just a page boundary, and, of course, GET will load files of any type, not just COM files.

Normal path searching is employed to locate the file. Thus if the SCANCUR option is off and the path does not explicitly include the current directory, the file will not be loaded from the current directory unless a colon is placed before the file name. Explicit directories can always be named using either the DU or DIR forms.

3.4.1. GET and Memory Protection

The GET command can have memory protection enabled or disabled depending on the setting of the FULLGET option in Z33HDR.LIB. If protection is enabled (FULLGET off), loading over any part of the operating system, including resident system extensions below the command processor or page zero of memory, will be prevented, and the error handler will be invoked. For an attempt to load to page zero of memory, the error will be a bad-number error; for an attempt to load above the top of the TPA, the error will be a TPA-overflow error. If memory protection is disabled, a file can be loaded to any address whatever, even if that results in destruction of the operating system. Sophisticated users will probably prefer to have this additional power and will accept the responsibility that comes with it.

If the address expression is not valid (not a number, for example), if the specified file is ambiguous (contains wild card characters), or if the specified file cannot be found along the search path, error handling is engaged so that the user can correct or abort the command as he wishes.

3.5. GO

The GO command has not changed at all. However, to avoid any possible confusion, we will state explicitly here that GO always calls (runs) the code at address 100H, even if the most recently loaded transient program had a type-3 environment and was loaded at an address other than 100H. Thus GO re-executes the most recently loaded type-1, type-2, or conventional CP/M transient program. It is equivalent to the command "JUMP 100".

3.6. JUMP

The JUMP command in Z33 has been enhanced to provide the full functionality of the GO command but with the additional capability of specifying the address to call. In Z30, JUMP could not be used with parameters in the command tail, because the first parameter was always the address number. With the Z33 JUMP command, after the address is evaluated, the command tail is reparsed beginning with the token after the address. Thus the command

```
JUMP 100 TOKEN1 TOKEN2 REST-OF-COMMAND-TAIL
```

functions identically to the command

```
GO TOKEN1 TOKEN2 REST-OF-COMMAND-TAIL
```

JUMP can be used to extend Bruce Morgen's POKE&GO technique to files with a type-3 environment. Suppose a program called MYPROG has been linked with a type-3 environment for execution at address 4000H and that we wish to execute "MYPROG SOURCE DESTINATION" after changing a configuration byte at offset ODH in the file to 0FFH. We can use the following command sequence:

```
GET 4000 MYPROG.COM
POKE 400D FF
JUMP 4000 SOURCE DESTINATION
```

This could be automated, for example, using an alias called perhaps MYPROG1 with the following script:

```
GET 4000 MYPROG.COM
POKE 400D FF
JUMP 4000 $*
```

One could then simply issue the command MYPROG1 SOURCE DESTINATION.

3.6.1. A Possible Rare Problem with JUMP, SAVE, and the ECP

There is one less than ideal aspect to the JUMP command that could cause a problem in rare instances, and we want to explain the problem here so that you will not be surprised if it ever happens to you. Suppose we enter the following command sequence to run our type-3 directory-display program DIR first on one directory and then on a second password-protected directory:

```
DIR PUBLIC:*.COM
JUMP 8000 PRIVATE:*.COM
```

When the JUMP command executes, we will be asked for the password to the PRIVATE directory TWICE. During the first parse, the tokens '8000" and 'PRIVATE:*.COM" will be parsed. When the line is reparsed with the pointer at the second token, the tokens "PRIVATE:*.COM" and "" (null token) will be parsed. Since 'PRIVATE:' appears again, the code will ask for the password again. Since the code to deal with this would probably be somewhat lengthy, since the situation will arise rarely enough, and since it is a problem that one can live with, we have left it this way.

This same problem can occur with the SAVE command for the same reason. It can also occur with extended command processing for a similar reason. When extended command processing (using the ECP) is initiated, the command line is reparsed using the command as the first token of the tail and the first token as the second token. If the formerly first token contains a reference to a passworded directory, then you are asked for the password a second time.

3.7. REN

The REN command normally functions just as it did in Z30. However, improved error handling has been provided, and one serious defect has been corrected. Consider the command line

```
REN PROG.COM=NEWPROG.COM
```

Suppose that PROG.COM already exists and that NEWPROG.COM, unexpectedly, does not. In Z30 one would first be prompted for deletion of PROG.COM. Only after it had been deleted would one be told that NEWPROG.COM was not found! In Z33, the check for the existence of the source file is performed first.

If either the source or destination file expression is ambiguous (wildcards) or if the source file does not exist, error handling is engaged so that the user can correct the command or abort it as desired. If the destination file already exists, the user will be prompted for its deletion. Since ZEX input redirection is disabled for all resident commands, the user will be able to supply the answer to the prompt. The console bell will ring at this unexpected prompt (unless the BELLFL option is set off).

3.8. SAVE

The Z33 SAVE command finally operates as described in ZCPR3: The Manual. The 'S' option described there never existed before. Here are some example commands:

```
SAVE 15 FILE1.TYP   Saves 15 decimal pages (30 records of 128
                    bytes each) to FILEl.TYP
SAVE 15H FILE2.TYP  Saves 15 hex - 21 decimal pages (42 records)
                    to FILE2.TYP
SAVE 15 FILE3.TYP S      Save 15 decimal records (7.5 pages) to
                    FILE3.TYP
SAVE 15H FILE4.TYP S Saves 15 hex - 21 decimal records (10.5 pages)
                    to FILE4.TYP
```

If the number expression is invalid (not a number at all, not a proper number in that radix, or too large a number), if the file specification is ambiguous, or if the disk or disk directory becomes full, error handling is engaged so that the user can attempt to recover or can abort the command as desired.

If the designated file already exists, the user is prompted for its deletion. ZEX input redirection is turned off during the prompt so that the user will be able to supply the answer. The bell will ring at this unexpected prompt (unless the BELLFL option is set off).

The character used for the sector option ('S' above) is configurable in Z33HDR.LIB using the SECTCH equate. As with the ERA command, if this

character is set to ' ' (space character), then any character at all in the third token (for example, 'R' for record as well as 'S' for sector) will invoke the sector option.

3.9. TYPE/LIST

These commands generally function as before. However, if the file name is ambiguous or if the specified file does not exist, then error handling is engaged, allowing the user to correct or abort the command as desired.

The paging toggle character, as with the ERA and SAVE commands, can be set to ' ' (space character) to allow the presence of any option character to toggle the paging mode.

The number of lines to use per page is taken from the data in the environment. Since the code required to determine which CRT is in effect is lengthy, the TYPE command in Z33 simply uses the number of text lines specified for CRT0. This costs only four bytes over using the fixed value that was used in Z30. We envision elimination in the future of multiple CRT and printer definitions in the environment. Instead of selecting among the two CRTs and four printers, a future utility could simply load data into single definitions for each device. This would free up space in the environment for other information and would give the user unlimited flexibility in defining the devices.

4. INSTALLATION

In this section we will describe the procedures used to install Z33 in a system currently running a version of Z30. As you will see, the procedure is only slightly technical and not actually difficult. However, with the large variety of computer systems around, procedures can vary widely, and we will not be able to give a prescription that will work in all cases.

4.1. Types of Existing Installations

We will distinguish two major types of ZCPR3 systems in use -- manually installed and automatically installed systems. The procedures for converting to Z33 are quite different in principle, though less so in practice. The original method for implementing ZCPR3 required that the user relocate the BIOS and BDOS (or ZRDOS) components of the operating system lower in memory to make room above the BIOS for the buffers required by ZCPR3. Late in 1984 Echelon introduced the first of its ingenious auto-install versions. It was called Z3-Dot-Com because the computer's standard CP/M was converted instantly to a ZCPR3 system simply by executing a file called Z3.COM. In January, 1985, a more advanced auto-install version, called Z-Com, was introduced. it included ZRDOS, the new Echelon replacement for the BDOS disk operating system, and it supported IOPs (input/output packages). Although Z3-Dot-Com was superseded by Z-Com and is no longer actively sold, many existing systems utilize it.

4.2. Installation Methodology

There are four basic steps to the installation process: 1) gathering the necessary files; 2) selecting the options desired; 3) assembling the new command processor module; and 4) installing the new command processor into the system. We will now discuss each of these steps. Only the last one is different for auto-install and manual install versions.

4.3. Collecting the Files

Collecting the files is rather straightforward, since all but one of the files needed should have been received in one package. The standard distribution Z33 system includes, among others, the following files:

ZCPR33.Z80	The main command processor source code
Z33MAC.LIB	Macro definitions for ZCPR33
Z33HDR.LIB	Configuration file

The one additional file that you must supply is Z3BASE.LIB, the file that describes the memory allocation in your system. If your present version of ZCPR3 was manually installed, you should already have this file. If not, you will have to create one. A complete discussion of that subject is beyond the scope of this guide, and you should consult ZCPR3: The Manual for further information. We can offer the following hints, however. If you run the utilities SHOW and Z3LOC, between them they will reveal all of the data you

need. Take a generic version of Z3BASE.LIB from the original Z30 release and edit it to reflect the addresses appropriate for your system.

If you are using an auto-install version, there is a generic Z3BASE.LIB file that comes with the package. There are two changes that have to be made. First you have to determine the address of your environment descriptor. This can be done in several ways. You can run the SHOW command or the Z3LOC command with the 'Z' option. Alternatively, you can load an installed ZCPR3 program into memory using "GET 100 PROGRAM.COM" and then peek at addresses 109H and 10AH to get the environment address. You have to put this value into the Z3ENV equate at the beginning of the Z3BASE.LIB file (also, if the symbol Z3ENV is defined with a SET directive, 'SET' should be changed to 'EQU' or 'DEFL' to be acceptable to a Zilog-mnemonic assembler).

The second change is required to correct an oversight in the distributed file. You must add a line to define the address of the CCP. This line, which can be placed right after the Z3ENV equate, should read

```
ccp equ z3env - 1a00h
```

Your Z3BASE.LIB file should now be ready for use.

4.4. Choosing the Options

Now you have to edit the Z33HDR.LIB file to select the features to include in your personal implementation of Z33. There being so many options to choose among, this may well be the hardest step of all. Because the command processor is limited in size to no more than 2K (2048) bytes, not all features can be included. Elsewhere in this manual and, to some extent, in the Z33HDR.LIB file itself, we have given overall guidelines to follow in selecting features. One of these is to begin by choosing the command processing options and only then to add resident commands as space permits (that's why we put the resident command equates at the top of the HDR file). The section on resident commands makes suggestions as to which commands should be chosen first for inclusion. If, as we are sure many will be, you are too impatient to read the other sections of this guide carefully before trying to install Z33, extensive comments have been included in Z33HDR.LIB to make it to a high degree self-explanatory.

The main consideration that will affect your choices is whether the system is to be open or secure. The distribution Z33 package includes several example Z33HDR.LIB files, some for open systems and some for secure systems. If your system will run in secure mode some of the time and open mode at other times, we recommend using one of the secure versions with the options for setting aside security when the wheel byte is set.

4.5. Assembling the Command Processor

Once the files have been collected and Z33HDR.LIB has been edited, the next step is to assemble ZCPR33.Z80. The Z33 source files have been prepared specifically for Echelon's ZAS assembler, and we will first describe how to use ZAS to produce the required object file. Later we will have a few words to say about assembling ZCPR33 with other assemblers.

ZCPR33.Z80 can be assembled to either of two types of object file: absolute or relocatable. In an absolute file, either of type HEX or COM, all of the addresses have been defined and coded in. With a relocatable file, with file type REL, the address at which the file will run has not yet been coded in. Instead, the file includes additional information that allows it to be converted during a subsequent linkage step to run at whatever address is later specified.

4.5.1. Assembling Z33 with ZAS

From our experience, installation of the command processor on most computers can be accomplished using an absolute object file. Some systems, however, use a relocatable (REL) object file. You may have to read ahead in the next section to know which type of file will be required for your system. If you have a choice, our recommendation is that you use an absolute COM file. With Echelon's ZAS assembler in its normal configuration (default output file in REL-format), the following command line will generate a relocatable output file named ZCPR33.REL:

```
ZAS ZCPR33
```

If you are going to generate relocatable output, the REL equate at the beginning of Z33HDR.LIB must be set to true, otherwise the file will include an expression that is not a valid relocatable item, and it will include an ORG statement with a specific address for the code. This is contradictory to the nature of a relocatable file.

To generate an absolute file named ZCPR33.HEX, use the 'H' switch on the command line as follows:

```
ZAS ZCPR33 H
```

If you are going to generate absolute output, the REL equate at the beginning of Z33HDR.LIB MUST be set to false. Otherwise there will be no ORG statement included in the code to specify the starting address, and the output file will be for a command processor running at the beginning of the TPA.

If you have chosen the options in Z33HDR.LIB properly, the file will assemble without errors to either type of file. Unfortunately, if the code is too long, you will get an assembly error message and will have to disable some options. If it is significantly smaller than 2048 bytes (for absolute files you will learn this later, as explained below), then you can try adding some more options. The Z33HDR.LIB file should be adjusted until you have a system that 1) has the features you want and 2) fits within the 2K allocation.

If you are vorking with an absolute file, you should now convert the HEX file to a COM file using MLOAD and the following command line:

```
MLOAD ZCPR33
```

Don't be surprised when the console bell beeps and you get the following warning message:

```
++ Warning: program origin is NOT at 100H ++
```

This message is expected, since the origin is at the address of the command processor. MLOAD will also give you valuable additional information, including the number of bytes loaded in both decimal and hex notation (do not confuse this with the number of bytes written to the output file, which will be rounded up to the next whole record). At this point you can determine if there is likely to be room for more options. One other warning is in order. Although ZCPR33.COM is a COM file, you cannot run it; it is almost guaranteed to crash your system if you try. If this makes you too nervous, you can rename it to something else like ZCPR33.BIN (binary image file). Just be sure to use the proper name in the instructions below.

4.5.2. Assembling Z33 with Other Assemblers

Before we go on to the next step in the installation, let us touch briefly on the question of other assemblers.
The only other assemblers that are known to assemble ZCPR33.Z80 in its distributed form with no modifications (other than Echelon's ZAS) are the SLR assemblers (Z80ASM, SLR180, or the virtual versions of them). With these assemblers, you should choose the option to generate the ZCPR33.COM file directly.

Any assembler that can process the Zilog mnemonics used in the source code and has a macro capability can probably be used, with more or less editing of the Z33 source code. One item likely to cause errors is the use of the square brackets ("[" and "]") required by ZAS instead of parentheses as used by other assemblers. So if you want to use another assembler, you will have to edit the source files (ZCPR33.Z80, Z33MAC.LIB, and Z33HDR.LIB) and replace square brackets by parentheses. This can easily be done with standard search-and-replace operations available in most editors. Also, a potential subtle problem is the usage of more than six characters in labels, particularly with the M80 assembler. Latest versions of M80 can handle long labels in the main code but not in macros, and the dummy parameters used in the macros in Z33MAC.LIB have to be shortened. One other change is required with M80: the MACLIB lines in ZCPR33.Z80 must be made uppercase. It is possible that other changes would be required if the assembler does not support the full set of opcodes, conditional expressions, and macro constructs. Use any error messages generated as a clue.

It is worth noting that it would be a very time-consuming task to attempt to adapt the Z33 source to a non-macro or 8080-mnemonic assembler such as ASM, MAC, etc. However, if you have no choice, good luck!

4.6. Installing the Command Processor into the System

We now turn to the final step -- installing the new command processor module into the operating system image. One thing you should definitely do now before you proceed any further (if you have not already had the prudence and good sense to do it) is make a new system disk to experiment with. Do not try to install the new system on your only complete copy of your current system! Duplicate the disk, and perform your experiments on the copy.

Manually installed ZCPR3 systems may require any of a number of possible installation procedures. Before we plunge into that complex subject, we will first describe the procedure used with the auto-install packages. Many users

of Z-Com have expressed concern that Z33 would not be installable in their systems. On the contrary, it can be installed, and the procedure is extremely easy.

4.6.1. Installation in Z-Com and Z3-Dot-Com Systems

An auto-install system has one very significant advantage over a manual install system -- the new command processor can be tested without even installing it! The auto-install versions of ZCPR3 do not load the command processor from the system tracks of the disk. Instead, the command processor is kept in a file. If you look in directory A15 you will find it. In the case of Z-Com, the file is called ZC.CP (Z-Com Command Processor); for Z3-Dot-Com it is called Z3.SYS (Z3.COM System). To test your new Z33 command processor, all you have to do is rename ZC.CP or Z3.SYS to some other name (in case you want it back later) and copy the ZCPR33.COM that MLOAD produced to directory A15 under the name ZC.CP or Z3.SYS. Then warmboot the system with a control-C, and, presto, the new command processor is running. You can now experiment with it and determine whether or not you are satisfied with the features you selected.

Once you have a command processor with which you are satisfied, only one additional simple step is required to make the auto-install system 'boot up' immediately with the new command processor (before, it only appeared after the first warmboot). When the Z-Com or Z3-Dot-Com system is booted by entering the commands 'ZC' or 'Z3', respectively, the command processor is loaded from ZC.COM or Z3.COM, where it resides beginning at address 200H. The simplest way to patch in the new command processor is with the following command sequences (you could make an alias to automate this):

```
For Z-Com:
     GET 100 ZC.COM      Load ZC.COM into the TPA
     GET 200 A15:ZC.CP   Overlay the new CPR
     SAVE 45 ZCNEW.COM   Save the new version

For Z3-Dot-Com:
     GET 100 Z3.COM      Load Z3.COM into the TPA
     GET 200 A15:Z3.SYS  Overlay the new CPR
     SAVE 39 Z3NEW.COM   Save the new version
```

To test the new versions, return to standard CP/M using the exit command ('ZCX' or 'Z3X') and then reinvoke ZCPR3 using the new loader. If it works, you can rename the file.

4.6.2. Installation into a Manual Install System

There are so many different ways in which operating system images are built and installed with different computers that we cannot possibly cover them all. Instead, we will try to describe two approaches that have fairly wide applicability. Both work from an absolute form of the command processor in a file called ZCPR33.COM. The installation methods based on relocatable system files are too machine-specific to cover here. You will have to follow the instructions that come with your computer or operating system.

4.6.2.1. Installation Using Disk Utility (DU3)

We will begin with a technique which we have not seen described before, though it is fairly obvious. If your system allows you to carry out these operations, this procedure is rather easy once you understand it, and it avoids a number of problems that can arise with more conventional installation techniques.

This procedure uses the disk utility program DU3. If you do not have a copy, get one. Put ZCPR33.COM in A0 and log into that directory. Invoke DU3, and use the following sequence of commands at the DU3 prompt:

```
T0 ; Select track 0 (first system track)
S1 ; Select sector 1 (first sector on track)
D  ; Display its contents on the screen
```

In almost all cases this sector contains code called the boot loader. What you have to do now is locate the command processor on the system tracks. Advance from sector to sector using the command

```
+D ; Go to next sector and display
```

until you see the Z30 command processor code. How do you recognize it? Two things will give it away. First of all, it begins with exactly two (and no more) identical jump instructions: C3 xx yy C3 xx yy. Secondly, in the text part of the display on the right, you will see the names of the resident commands, such as SAVE, GET, GO, and so on. Once you have found it, make note of the sector number; you will need it later.

Just for reference, here are the locations of the command processor on the system tracks of several popular computers for which this installation technique works. Do not take these values on faith -- check them. It is always possible that your machine will be different from the one used to ascertain this information.

Ampro and SB180	T0,S2
Televideo 803	T0,S3
Kaypro (normal)	T0,S2
Kaypro (TurboROM)	T0,S5
Morrow MD3	T0,S9

The next step is to load the new command processor image into what is called the queue. First you have to find the file. DU3 has the 'F' command especially for this purpose. Enter

```
FZCPR33.COM         Find file ZCPR33.COM
```

The display will be something like

```
DU3 A0? FZCPR33.COM
60 005A4350 52333320 20434F4D 00000010 |.ZCPR33 COM ...|
70 3A000000 00000000 00000000 00000000 |...............|
```

By looking at the number at the lower left, you learn which disk group the file ZCPR33.COM is stored in. In our example it is group 3A (hex).

There can be some complications at this point. If you find the following discussion too confusing, either use a different installation method, or ask someone (your local Z-Node sysop, for example) for help. The example above assumed that your disk had a group size of 2K or more. If your disk is a single-density floppy that uses 1K groups, there will be two numbers (e.g., 3A and 3C) instead of one. You will have to remember both of them. On the other hand, if you have very large disks containing many groups, it may take a word rather than a byte to contain the group number. For example, you might see a sequence like 3A 01. In this case the group number is 13A. If you are not sure which one it is, the steps described in the next paragraph will help confirm any guess you have to make.

Once you have determined the group number, go to that group and display its contents using the DU3 command sequence

```
Ggg  ; Go to the group (gg=3A in above example)
D    ; Display the contents
```

You should notice a strong resemblance between this display and the one you saw before when you located the command processor on the system track. Assuming you see something that looks like a command processor, you will now capture it in DU3's queue. Here is the command line to enter (it must be all one command, unlike the command sequences above that could be entered either singly or as a multiple command line using comma separators).

```
<B,+,*16
```

Here is what each part of the command does:

```
<B   ; Copy the disk data buffer into the queue
+    ; Advance to the next record
*16  ; Repeat this command sequence 16 times
```

This command copies 16 records or 2K bytes into the queue.
If your disks have 1K groups and you recorded two group numbers earlier, you would copy the file in two pieces into the buffer. The commands would be

```
Ggg1      ; Go to first group
<B,+,*8   ; Get first 8 records
Ggg2      ; Go to second group
<B,+,*8   ; Get the second 8 records
```

where gg1 and gg2 are the two group numbers (3A and 3C, for example).

Now that you have the command processor in the DU3 queue, all you have to do is load it into place in the system track. To do this, first return to the sector you identified earlier:

```
T0        ; Track 0
Sss       ; Go to sector determined earlier
D         ; Display it (just to make sure)
>B,W,+,*16; Copy queue onto disk
```

The last command, which again must be all on one line, accomplishes the following: >B ; Copy one record from queue to buffer
W ; Write the buffer to disk
+ ; Advance to the next disk record
*16 ; Repeat for 16 records

Now enter a control-C to warmboot out of DU3. The new command processor is installed and running!

4.6.2.2. Installation Using a SYSGEN Image

The most common system installation technique provided by the manufacturers of computers uses a program called SYSGEN (system generator) or something similar, such as TurboGen in the case of the Kaypro TurboROM system. We will suggest two possible ways to proceed if your computer uses this system. In both cases, you should have the ZCPR33.COM file in the currently logged in directory.

4.6.2.3. The Simpler SYSGEN Technique

The simpler of the two SYSGEN techniques proceeds according to the following four steps.

1. Run the SYSGEN program. When asked for the source drive, enter A followed by carriage returns. When asked for the destination drive, do not answer with any drive letter. Just hit a carriage return to exit from SYSGEN. The system image for the operating system stored on the system tracks of the A disk is now in memory.

2. Use the 'P' (peek) command to find the beginning of the command processor code in the image. If your screen is not too fast, you might try entering the peek command as

```
P 800 4000
```

We have never seen a SYSGEN that puts the command processor at an address less than 800H. The command processor always begins on a sector boundary - the beginning address will always be xx80 or xx00. You want to try to recognize the first record of the CPR. The most easily recognized data structures are the names of the resident commands. The most common location for the CPR is 980H. Here are the values we found for several popular computers:

Ampro, SB180	980H
Kaypro (standard)	980H
Kaypro (TurboROM)	B00H
Morrow MD3	D00H
Televideo 803	980H

Again, do not take our word for it -- check the value yourself. It is always possible that your system has a different version of SYSGEN.

3. Now you have to install the new command processor into the system image. This is quite easy with the GET command:

```
GET aaa ZCPR33.COM
```

where 'aaa' is the hexadecimal address you determined in step 2 above. Now you have in memory the image of our new operating system.

4. For most versions of SYSGEN, all you have to do now is rerun the loaded memory image by issuing the command 'GO'. As before, you will be asked for the source disk. Don't answer the question this time; just enter a carriage return. Then you will be asked for the destination drive. Answer W, and follow that with carriage returns until you exit from SYSGEN. The new system is now running.

4.6.2.4. The More Complicated SYSGEN Technique

Some SYSGEN programs, such as the ones for the Ampro and SB180 computers, are too fancy. They refuse to allow you to rerun them with the system image already loaded in memory. They will use only an image that is stored in a disk file. So, we carry out the first three steps described above, but we then proceed slightly differently at step 4.

4. Now that you have the system image in memory, you have to save it in a disk file. Enter the following command

```
SAVE nn ZCPR33.SYS
```

where 'nn' is the size of the system image in pages (units of 256 decimal or 100 hexadecimal bytes). Now you are probably wondering what that value is! Here are several ways to deal with that question.

4a. This might be your lucky day, and SYSGEN may have reported on exit the number of pages you should save. All SYSGEN programs should do this, but we don't recall ever having seen one that does.

4b. You can sidestep the question entirely and just enter a number that you are sure is too big. That will waste some disk space temporarily, but it will work just fine. A number like 48H or 72 (decimal, 18K bytes) certainly should be enough.

4c. You could try reading the documentation that came with your computer (a novel idea, we are sure). Again, the information should be there, but, again, it probably is not.

4d. You can try to get the information by running MOVCPM, the other system generation program that generally comes with computer systems. We have more often seen MOVCPM programs that report the number of pages to save. Of course, if you want to use this technique, it will wipe out the image you have constructed in memory, and you will have to start over again from the beginning.

4e. Finally, if you are really fastidious and want to know the exact number, you can do the following. Before performing steps 1, 2, and 3 described previously, first run a debugger, such as DDT, ZDM, or DSD,

to initialize a large block of memory to zero. First use the 'F' (fill) command:

```
F100,6000,0    ; Zero memory from 100H to 6000H
```

Then exit from the debugger and follow the three installation steps described previously. Now use the 'P' (peek) command again to find where the system image in memory ends. Specifically, scan to find the page at which memory that still contains zeros begins. Subtract one from that value and you have the number required for step 4 above. For example, if the system image ended (the zeros began) at address 2980H, the first full page of initialized memory would be 3000H, and the number of pages to save would be 29H (or 41 if you prefer to enter a decimal number, though SAVE will take 29H). Don't forget that you are working with hexadecimal numbers, so if the address had been 3000H, the number of pages would be 2FH (30H-1=2FH).

5. Now that you have saved the new system image to a file called ZCPR33.SYS, you can run SYSGEN again to install it onto the disk. Enter the command

```
SYSGEN ZCPR33.SYS
```

You will be prompted for a destination drive. Enter 'A' and some carriage returns. Your new system is now running.

Bibliography

Title	Author	Publisher
ZCPR3: The Manual	Richard Conn	New York Zoetrope
ZCPR3: The Libraries	Richard Conn	Echelon
ZCPR3 and IOPs	Richard Conn	Echelon
Z-System User's Guide	Bruce Morgen Richard Jacobson	Echelon
ZRDOS Programmer's Guide	Dennis Wright	Echelon
Z-NEWS (newsletter)		Echelon
The Programmer's CP/M Handbook	Andy Johnson-Laird	Osborne/McGraw/Hill
Programing the Z80	Rodney Zaks	Sybex

This Page Left Blank

I N D E X