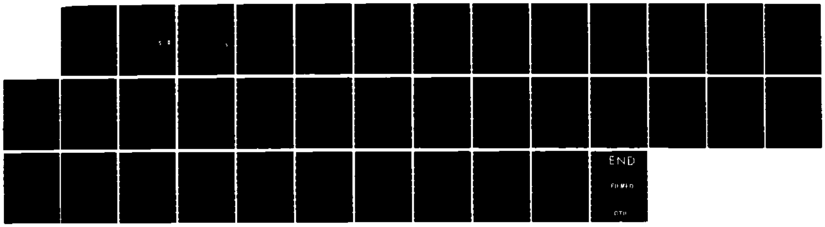
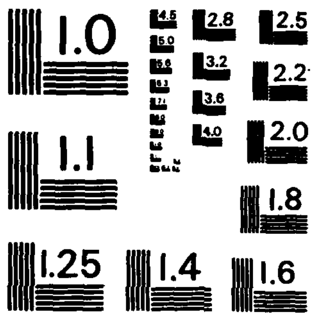
AD-A162 420   CONVERS 33 - A THREADED CODE INTERPRETIVE COMPILER FOR   1/1
THE PDP-11 MICROCOMPUTER(U) ARIZONA UNIV TUCSON DEPT OF
CHEMISTRY G R SIMS ET AL. 25 NOV 85 TR-41

UNCLASSIFIED   N00014-83-K-020   F/G 9/2   NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS - 1963 - A

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 41	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CONVERS 3.3 -- A Threaded Code Interpretive Compiler for the PDP-11 Microcomputer		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) G. R. Sims and M. B. Denton		8. CONTRACT OR GRANT NUMBER(s) N00014-83-K-0268
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Chemistry University of Arizona Tucson, Arizona 35721		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 051-549
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE November 25, 1985
		13. NUMBER OF PAGES 33
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale: its distribution is unlimited.

DTIC ELECTE DEC 18 1985 S D
D

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Prepared for publication in the Journal of Chemical Information and Computer Sciences

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Threaded code, interpretive compiler

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A threaded code interpretive compiler called CONVERS 3.3 is presented. This language is intended for use in laboratory instrument control and data acquisition applications. Discussed are the philosophy behind the development of this language, the internal operation of the interpreter and the compiler, and the advantages this language offers over other programming languages in its intended applications. CONVERS is compared and contrasted to BASIC interpreters, FORTRAN compilers, and particularly to typical FORTH interpretive compilers. Programming examples and execution time benchmarks for code written in BASIC, FORTH, FORTRAN, and CONVERS are presented.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

OFFICE OF NAVAL RESEARCH

Contract N00014-83-K-0268

Task No. NR 051-549

TECHNICAL REPORT NO. 41

CONVERS 3.3 -- A Threaded Code Interpretive Compiler for the PDP-11 Microcomputer

by

G.R. Sims and M.B. Denton

Prepared for Publication in the Journal of Chemical Information and Computer Sciences

University of Arizona
Department of Chemistry
Tucson, Arizona 85721

November 25, 1985

CONVERS 3.3 -- A THREADED CODE INTERPRETIVE COMPILER FOR THE PDP-11 MICROCOMPUTER

G. R. Sims and M. B. Denton
Department of Chemistry
University of Arizona
Tucson, AZ 85721

Abstract

A threaded code interpretive compiler called CONVERS 3.3 is presented. This language is intended for use in laboratory instrument control and data acquisition applications. Discussed are the philosophy behind the development of this language, the internal operation of the interpreter and the compiler, and the advantages this language offers over other programming languages in its intended applications. CONVERS is compared and contrasted to BASIC interpreters, FORTRAN compilers, and particularly to typical FORTH interpretive compilers. Programming examples and execution time benchmarks for code written in BASIC, FORTH, FORTRAN, and CONVERS are presented.

CONVERS VERSION 3.3A FOR THE PDP-11

CONVERS was developed in the laboratories of the Department of Chemistry at the University of Arizona in response to the need for an interactive programming language optimized for instrument control and data acquisition. Many of the ideas imbedded in CONVERS were first implemented in FORTH, a language developed by Chuck Moore, then of the National Radio Observatory. The original authors of CONVERS, S.B. Tilden and M.B. Denton, felt that while FORTH is a brilliant concept for instrumentation control, it also has some real shortcomings. Some of the problems with original FORTH are;

(1) difficult to tailor the system for different peripherals,
(2) almost no error checking,
(3) very difficult to understand the "inner workings".

When CONVERS version 2.1 for the INTEL 8080 based microcomputers was written, it remedied many of these drawbacks. CONVERS is much more forgiving than FORTH when an error is made. It is also very simple to modify CONVERS to communicate with peripheral devices, no mysterious system generation process is involved. CONVERS 2.1 for the INTEL 8080 has been in use for many years now and has found wide application in academic and industrial environments. The success of CONVERS is reflected in the fact that many FORTH systems written in recent years have begun to resemble CONVERS more than original FORTH.

CONVERS version 3.3A for the PDP-11 departs from original FORTH even more than CONVERS 2.1. The changes implemented are mainly due to the rapid advancement in microcomputer hardware over the past 5 years. Whereas micros in the past were typically equipped with 8K or 16K bytes of memory, todays micros are usually equipped with at least 64K bytes. Mass storage devices have also made tremendous gains, most 8080·type microcomputer systems were typically equipped with small single sided, single density 5 1/4 inch minifloppies. Today's microcomputer systems often have several megabyte storage on Winchester and double sided, double density floppy disk drives.

CONVERS version 3.3A for the PDP-11 capitalizes on these improvements in hardware by including descriptive error messages and more thorough error checking. A directory system for the disk is also implemented in order to simplify the writing of source code and make CONVERS program and data storage compatible with the RT-11 operating system. The entire initial machine code dictionary has undergone careful re-evaluation and rewriting to optimize for usefulness and speed.

To keep CONVERS understandable by those who are not professional programmers, the initial machine code dictionary is written in as simple form as possible. All entries are coded in assembly language. The use of macros has been avoided. This means anyone with a rudimentery understanding of assembly language and basic computer architecture can understand CONVERS at the most elementary level.

This manual was written to explain the basic concepts and programming techniques for the novice as well as to explain this version of CONVERS in detail for the experienced CONVERS programmer.

The first section of the manual, "CONVERS, an Interpretive Compiler" explains why CONVERS was created and why it is a unique language. The "CONVERS Primer" is a tutorial for those people who have never programmed in CONVERS or FORTH. The Unabridged Dictionary gives detailed information on all of the IMCD entries, and, finally, the appendixes contain detailed information on specialized subjects. This manual does assume that the user has at least a vague idea of how a computer works and what machine code is. Useful reference books to accompany this manual are the appropriate Digital processor manual, the *peripheral and interfaces manual*, and Machine and Assembly Language Programming for the PDP-11 by Gill.

CONVERS -- AN INTERPRETIVE COMPILER

The two common classes of high level languages are interpreters (BASIC, APL, LISP, etc.) and compilers (FORTRAN, PASCAL, "C", ALGOL, etc.). Each class has definite advantages and disadvantages. Interpreters are interactive, which makes them easy to learn and use, but the penalty for being interactive is slow execution time. Programs written in compiler languages generally run much faster, but compiler languages are more difficult to learn, and programs are very tedious to write and debug. One obvious question immediately arises -- why not incorporate the most desirable characteristics of compilers and interpreters into a single language? Additionally, due to the unique and diverse requirements found in many applications, why not allow the programmer the flexibility to actually tailor the language to fit a specific programming problem? Other requirements for this hypothetical language are high memory efficiency, fast execution time, and it must be easy to understand how the language works at the most elementary level.

A language described originally in 1974 by C.H. Moore, then of the National Radio Observatory, offers most of the desired characteristics described above. However, this language, called FORTH, has some major shortcomings. One major problem with FORTH is that its "internal workings" are exceedingly complex. The second major flaw is that FORTH has very few error messages and diagnostics. This is largely due to the fact that FORTH was conceived at a time when computer memory was very expensive.

Drawing from experience gained from programming in BASIC, FORTRAN, and FORTH, a new approach to software named CONVERS has been developed. The word "approach" has been used because CONVERS does NOT in any way resemble traditional programming languages. First of all, programming techniques are very different, being inherently "modular". Secondly, there is no separation of programming language and operating system.

In order to fully appreciate the CONVERS (and FORTH) "interpetive compiler" approach, it is first necessary to review the characteristics of typical interpreters and compilers.

A TYPICAL INTERPRETER

An interpreter is mainly a collection of commands (machine code subroutines) which can be executed upon request. Programming an interpreter simply involves requesting the execution of the appropriate commands in a logical order. A program enters the interpreter as symbolic source code which is input from either a terminal or mass-storage device. As the souce code is input, the interpreter accepts the commands one at a time, searches through its set of commands to see if the requested function exists, and if it does exist, the command is executed. If the requested function does not exist, or can not be executed as requested, the programmer is immediately informed.

The fundamental advantage of an interpreter over a compiler is that it is always much easier to learn to program, largely because the programmer is informed of a mistake as it is made. The ease of learning an interpeter language is evident by the popularity of BASIC in personal computers. In fact, BASIC was originally conceived as a "teaching language" for persons with absolutely no computer background (BASIC stands for Beginners All-purpose Symbolic Instruction Code). The prime disadvantage of an interpreter are low execution speeds. Interpreters are inherently slow because each command entered must be interpreted before it is executed. In most cases, the interpretation cycle takes much more time than actual command execution.

A TYPICAL COMPILER

A conventional compiler accepts symbolic source code from a mass storage device, interprets the commands, then transforms the commands into machine code which is then written to a mass storage device. The role of the compiler is only to translate the source code into machine (or "object") code. The compiler plays no role in the actual execution of the object code. To execute the program, the object code must first be loaded from the mass storage device.

The prime advantage of a compiler language is fast execution time. The compilation step requires a great deal of time, but the actual program execution speed is very fast since the program contains within itself all of the code it needs to execute. The big disadvantage of a compiler language is the non-interactive nature. To make any change in a program, whether it be to correct an error or make a modification, the entire source code must be recompiled, relinked, reloaded, and only then can the program be executed to see if the change was successful. This non-interactive approach makes writing programs very frustrating and time-consuming for an inexperienced programmer. The difficulty in learning a compiler language is reflected in the fact that after an entire college semester course in FORTRAN, students are still not considered to be experienced programmers.

THE CONVERS APPROACH

The prime advantage of an interpreter is its interactive nature. This is also the case with CONVERS. As soon as CONVERS is loaded and running, it is waiting to accept, interpret, and execute commands much like BASIC. Unlike traditional interpreters, however, CONVERS also has the ability to accept, interpret, and COMPILE symbolic source code. Thus CONVERS has two "states", the executive state where code is interpreted then executed, and the compile state where code is interpreted then compiled.

In both states, CONVERS is truly an interpreter, the difference being that it can be an "executive interpreter" or an "interpretive compiler".

The extremely powerful "interpretive compiler" mode is unique to CONVERS and FORTH. This is the mode that combines the best features of an interpreter and compiler and set CONVERS and FORTH apart as fundamentally better languages for their targeted applications. Since CONVERS is truly an interpreter, even while in the compile state, diagnostic error and warning messages continue to appear while compiling a program. Thus, CONVERS remains interactive while programs are being written!

The bulk of CONVERS is made up of small program units (subroutines) which have English language names. This collection of subroutines is called the "dictionary" and an individual subroutine is called a "dictionary entry". To execute a dictionary entry, the name of that entry is typed on the terminal. For instance, to execute the entry which rings the terminal bell, the word "BELL" is typed. Programming CONVERS involves the simple process of creating new dictionary entries. To put CONVERS in the compile state, a colon and space are typed, then the name for a new dictionary entry is typed. For instance;

```
: SOUND
```

creates a new dictionary entry called "SOUND". The entry "SOUND" can consist of any previously defined entry. Suppose for sake of argument that the entry "SOUND" should cause the terminal bell to ring five times. One way to achieve this is to type BELL five times;

```
: SOUND BELL BELL BELL BELL BELL ;
```

Now "SOUND" consists of the machine code required to ring the terminal bell five times. Typing the semicolon ends the new definition and returns CONVERS to the executive state. To execute "SOUND", its name is typed. It is important to reiterate here that the entry "SOUND" contains the actual machine code neccesary to ring the bell five times. When the "SOUND" entry is executed, CONVERS does NOT need to go through the time-consuming process of searching through the dictionary to find the "BELL" entry, the search step was done when "SOUND" was compiled.

The dictionary entries which make up CONVERS when it is first started (such as "BELL") are called the "INITIAL MACHINE CODE DICTIONARY" (hereafter known as the "IMCD"). When a new entry is programmed it becomes a part of the "USER DICTIONARY". CONVERS does not distingush between IMCD and the user dictionary entries, it is simply a term used to distinguish between what CONVERS consists of when first started and what a user adds to CONVERS while programming.

Since the entry "SOUND" is now a part of the dictionary, it can be compiled as part of yet another new entry such as;

```
: 1+2RING 1 2 + SOUND ;
```

The entry "1+2RING" consists of the machine code required to add the numbers 1 and 2, then ring the terminal bell five times.

One way to think of the dictionary is merely as an "erector set" of components which may be used over and over to assemble an ever increasingly sophisticated set of modules which can in turn be readily intermixed to achieve the desired final program. CONVERS does not have a limited command set because the act of programming actually creates new commands. Because new commands are actually created while programming, CONVERS can then be thought of as a language to write specialized languages! In a typical situation, the "language" that one would create will have the commands necessary to execute some task or series of tasks. The collection of user-defined entries designed to execute these tasks are called an application dictionary. An application dictionary is stored on a mass-storage device and loaded into CONVERS when needed. Once a task is finished, the user can remove an application dictionary (or the application dictionary can even remove itself). Typical application dictionaries might include floating point math operations, a text editor, data collection commands, etc., all written in CONVERS!

Since CONVERS is a true interpreter, it is very easy to learn. It takes most people two sessions with a terminal (4 to 6 hours) to learn the IMCD commands and the programming techniques. Actual coding and testing programs are almost trivial once a good algorithm has been established. Debugging is especially simple since each entry can be individually tested.

WHAT CONVERS SHOULD BE USED FOR

The interpretive compiler approach to programming languages is a general concept that is desirable in any computer language, but it is totally inconceivable to write a single language that is ideal for every imaginable application. CONVERS itself has been carefully optimized for use in control and data acquisition applications. CONVERS is NOT targeted for the types of applications that are performed on a mainframe computer; it is designed to operate on micro and minicomputers in a single-job, single-user environment. CONVERS is definitely the language of choice whenever a small computer system is interfaced to the "real world" through A/D converters, D/A converters, stepper motors, etc.

In general, all compilers and interpreters are too inflexible to use for any highly specialized application, such as instrument control and data acquisition since the only commands available to the programmer are those included in the language by the software vendor. The only way to add commands is an awkward process of writing the needed function in assembly language and forcing the high level language to call the function.

For years one of the most popular compilers, FORTRAN, has been misused for many applications. FORTRAN was intended to be strictly a "number crunching" language (FORTRAN is a hyphenation of FORMULA TRANSLATOR), and it is very good for this specific application. FORTRAN was NEVER intended to be used for the types of applications CONVERS is targeted for.

CONVERS, on the other hand, was never intended to be strictly a "number crunching" language; however, many simple data reduction and statistical calculations are as easy, or easier, to program in CONVERS than in other languages.

When numerical analysis of data is very complex (or if data reduction programs already exist in another language), CONVERS can be used for the instrument control and data collection process only. Later the data can be analyzed with programs written in a language designed for numerical analysis (such as FORTRAN). Thus, the appropriate language is chosen for the particular problem.

****

A CONVERS PRIMER

****

Since CONVERS is a interactive language it is best learned by spending some time at a terminal writing and executing dictionary entries. This primer is designed to help the beginning programmer learn CONVERS by explaining what the commonly used initial machine code entries (hereafter known as IMCD entries) do and how to use them in programming. Throughout the primer there are numerous examples which the student should try out as the primer text is being read. When an example is indicated, you type the underlined text and CONVERS will respond with the text within the < > symbols. For instance, in the example;

```
        1 1 + .    <2>
```

you type the "1 1 + ." and CONVERS will type the "2".

In addition to the examples are suggested problems the student should attempt to solve. Successfully solving the problems will indicate a thorough understanding of the IMCD entries presented in the examples. This primer does not describe all entries in the IMCD. Many entries are so rarely used that including them in this primer would only be a source of confusion. The unabridged dictionary listing contains detailed information on every IMCD entry. Once some programming experienced is gained, the unabridged listing should be examined in order to become acquainted with the full range of IMCD entries.

****

STARTING CONVERS

****

The file CONVER.SAV should be properly configured for the computer system on which it is to be run. Assuming it is, and CONVER.SAV is on the system disk, boot the RT-11 monitor and type;

```
        R CONVER
```

When CONVERS is running, it will respond with the message;

```
        <CONVERS VERSION 3.3A FOR LSI-11>
        <RX02 RESIDENT DISK HANDLER>
        <--RESET-->
        <--RECOVER-->
```

The resident disk handler message is dependent on the particular computer system it is running on. For now it is of no concern.

****

CONVERS TERMINAL INPUT AND OUTPUT

****

Because CONVERS is a highly interactive language, the most important and most often used input/output device is the console terminal. Certain terminal keys have special functions in CONVERS. These are;

RETURN or ENTER

A carriage return is a terminator. It signals that input is finished.

SPACE

A space is also a terminator. It has exactly the same function as a CR. This is very different from other languages where a space is a delimiter. A delimiter only separates words of input while a terminator indicates input is truly finished. CONVERS does not operate on strings of typed words, it operates on each word as it is typed in.

^C

Typing a ^C executes the RESET entry. This causes any executing program to be unconditionally terminated, the system input and output device is reset to the console terminal, and most system variables are reset to the same conditions as when CONVERS was started. The ^C key might be called the "panic button".

^S

Typing a ^S key causes the processor to hang in a loop until a ^Q or ^C is typed. All processing is stopped except for servicing interrupts.

^Q

Typing a ^Q breaks the processor out of the hang loop discussed above.

RUB or DEL

Typing a RUB deletes a previously typed character. If CONVERS is configured for a video terminal, the character deleted is removed from the screen. If CONVERS is configured for a printing terminal, a "*" is typed to indicate a character was deleted.

All other keys, including escape and control keys are assumed to be part of an entry name or a number. CONVERS treats upper and lower case letters equally.

****

SECTION I -- NUMBER INPUT/OUTPUT AND RADIX CONTROL

****

Number input is done simply by typing a number on the terminal. After a space or return is struck, the typed number will be converted from ASCII to binary and placed on top of the stack. A number can be entered in either base 10 (decimal radix) or base 8 (octal radix). When CONVERS is first started, it is in decimal radix. Number output is also very simple. When a period is typed, followed by a terminator (space or CR), the top number on the stack is converted from binary to ASCII and output to the terminal. When a number is output to the terminal, it is removed from the stack. Of course, there must be a number on the stack to output before a "." is typed. If the stack is empty and a "." is typed, the error message;

```
STACK UNDERFLOW
--RECOVER--
```

will appear.
Changing number input/output radix between decimal and octal is very simple as it only involved typing the words OCTAL or DECIMAL;

OCTAL

Forces number input/output to be done in octal radix. Octal is a high precedence entry so the number radix can be changed while in the middle of a colon definition.

DECIMAL

Forces number input/output to be done in decimal radix. DECIMAL is a high precedence entry so the number radixcan be changed while in the middle of a colon definition.

The OCTAL and DECIMAL entries are known as "permanent" radix changes because if the word OCTAL is typed, all number input and output is done in octal radix until a DECIMAL is typed. If CONVERS is in octal radix and a single number in decimal radix needs to be entered, type a D and a space, then the number. Likewise if a single number needs to be output in decimal radix, type a "D.". The entries "D" and "D." are known as temporary radix changes because they are in effect for only one number. Similar temporary radix change entries exist for octal radix also;

D

Input a single number in decimal radix.
EXAMPLE: <u>OCTAL 34 D 34 .</u> <42> <u>.</u> <34>
The first 34 is entered in octal radix and the second is entered in decimal radix. Both numbers are output in octal radix.

O

Input a single number in octal radix.
EXAMPLE: DECIMAL O 34 .   <28>

D.

Output a single number in decimal radix.
EXAMPLE: OCTAL 20 20 .   <20> D.   <16>

O.

Output a single number in octal radix.
EXAMPLE: DECIMAL 16 16 .   <16> O.   <20>

Since the PDP-11 is a 16 bit computer, the largest integer number that can be entered in octal radix is 177777. The largest number that can be entered in decimal radix is 65535. In decimal radix, bit 15 (the most significant bit) of the processor is usually interpreted as a sign bit. If bit 15 is set, the number is negative, and if bit 15 is clear, the number is positive. When bit 15 is treated as the sign bit, the range of integer numbers that CONVERS can accept is -32768 to +32768. When the entry DECIMAL is typed, the number output is "signed", that is, the numbers are output in the range from -32768 to +32768. Another entry exists however, which outputs numbers in "unsigned" format, from 0 to 65535;

/DECIMAL/

Forces number input/output to be done in decimal radix, but the output is unsigned.

Whether DECIMAL or /DECIMAL/ is the number radix, the number 65535 is the largest number CONVERS will accept. The only change is in the output format;

EXAMPLE: DECIMAL 65535 . <   -1>

EXAMPLE: /DECIMAL/ 65535 . <65535>

If a number larger than 65535 (decimal) or 177777 (octal) is typed on the terminal, the error message;

NUMBER TOO LARGE

will appear. The only characters that may be typed when entering a number in decimal radix are the numerals 0 through 9, and the minus sign (-). In octal radix, the only characters that may be typed are the numerals 0 through 7. If any illegal character is typed, the error message;

NO SUCH ENTRY

will appear.

****

SECTION II -- TEXT OUTPUT

****

It is essential that an interactive computer language be able to output messages to a programmer or program operator. Several entries exist which output text, and/or aid on formatting number output.

CRLF

Outputs a carriage return and line feed to the terminal.

SPACE

Outputs a space to the terminal.

BELL

Rings the bell on the terminal.

OUTPUT

Fetches the top number on the stack and interprets it as the starting address of an ASCII string. The ASCII string is output to the terminal until a zero byte is detected.

TEXT

Allows the programmer to define an entry that, when executed will put the address of a stored ASCII string on the stack.
EXAMPLE: TEXT HOWDY-MESSAGE HOWDY! ^
An entry called HOWDY-MESSAGE has been created. When this entry is executed, the address of the ASCII string "HOWDY!" is put on the stack. This string can then be output with the OUTPUT entry.
EXAMPLE: HOWDY-MESSAGE OUTPUT <HOWDY!>
The TEXT entry will store any ASCII character including CRLF. The up arrow character (^) is used to indicate end of text. There is no limit to the number of characters that may be stored within a string.

SECTION III -- ARITHMETIC AND LOGIC OPERATIONS

In CONVERS, all logical and arithmetic operations are performed on the stack, in reverse polish notation, in the same manner as Hewlett-Packard calculators. To perform an arithmetic or logic operation, the parameters used in the operation must first be placed on the stack. During execution of the operation, the parameters are removed from the stack. After the operation is completed, the result is left on the stack.

As an example, consider the addition entry, "+" which adds the top two numbers on the stack. To add the number 13 and 47, the following is typed;

```
        13 47 +
```

When the "+" executes, the 13 and 47 are removed from the stack. The only thing left on the stack is the result, 60.

The stack logic and arithmetic entries are fundamental to any CONVERS program; therefore, they should be memorized. This is not really very difficult since there are not very many entries (41 total) and most entries have names which are familiar, such as "AND", "OR" "/" etc. Listed below are the rest of the stack operators. Type in the examples and make sure you are fully acquainted with all details of these entries.

```
+
        Add the top 2 numbers on the stack.
        EXAMPLE: 14 37 + .   <60>


-
        Subtract the top 2 numbers on the stack.
        EXAMPLE: 24 13 - .  <11>   (decimal radix)


INC
        Increments the top number on the stack by one. This is
        equivalent to 1 + except it uses less memory and executes
        faster.
        EXAMPLE: 1 INC .   <2>
        EXAMPLE: -2000 INC . <- 1999>


DEC
        Decrements the top number on the stack by one. This is
        equivalent to 1 - except it uses less memory and executes
        faster.
        EXAMPLE: 1 DEC .   <0>
        EXAMPLE: -2000 DEC . <- 2001>
```

*

Multiply the top two numbers on the stack. The multiplication is unsigned and no overflow check is made.
EXAMPLE: 200 10 * . <2000> (decimal radix)
EXAMPLE: -134 -786 * . <-25748> (decimal radix)
EXAMPLE: 20000 3 * . <-5536> (overflow to sign bit)
EXAMPLE: 20000 4 * . <14464> (overflow to carry bit)

/

Divide the top two numbers on the stack. All divisions are signed.
EXAMPLE: 13 4 / . <3>
EXAMPLE: -30000 3 / <-10000>
EXAMPLE: 34 79 / . <0>

/R

Similar to / except the remainder is also provided. The top number on the stack is the quotient and the second number is the remainder.
EXAMPLE: 13 4 /R . <3> . <1>
EXAMPLE: 793 796 /R . <0> . <793>

/A

Similar to / except the quotient is rounded off rather than truncated.
EXAMPLE: 13 4 /A . <3>
EXAMPLE: 15 4 /A . <4>

2/

Quick divide by two. This shifts the data bits in the top number on the stack right one bit. This is equivalent to an arithmetic shift right. The sign bit is not shifted so the division is signed. "2/" will use less memory and execute faster than "2 /", but it accomplishes the same function.
EXAMPLE: 175000 2/ . <176400> (octal radix)
EXAMPLE: -1536 2/ . <- 768> (decimal radix)

2*

Quick multiply by two. This shifts the data bits in the top number on the stack left one bit. This is equivalent to an arithmetic shift left. The sign bit is not shifted so the multiplication is signed. "2*" will use less memory and execute faster than "2 *", but it accomplishes the same function.
EXAMPLE: 245 2* . <490> (decimal radix)

/2//

Quick unsigned divide by two. This shifts all bits, including bit 15, right once. This is equivalent to a rotate right after

the carry bit is cleared. "/2//" is the only way to perform an unsigned division, i.e., a division where bit 15 is considered to be a data bit.
EXAMPLE: 176500 /2// . <77240> (octal radix)
EXAMPLE: -704 /2// . <32416> (decimal radix)

/2*/

Quick unsigned multiply by two. This shifts all bits, including bit 15, left once. This is equivalent to a rotate left after the carry bit is cleared.
EXAMPLE: 76543 /2*/ . <175306> (octal radix)
EXAMPLE: -246 /2*/ . <-492> (decimal radix)

ABS

Take the absolute value of the top number on the stack.
EXAMPLE: -1024 ABS . <1024> (decimal radix)
EXAMPLE: 1024 ABS . <1024> (decimal radix)
EXAMPLE: 177777 ABS . <1> (octal radix)

NEG

Negate the top number on the stack by taking the 2's complement. If the number on the stack is positive, the NEG entry will make it negative. If the top number on the stack is negative, the NEG entry will make it positive.
EXAMPLE: 34 NEG . <-34> (decimal radix)
EXAMPLE: -34 NEG . <34> (decimal radix)

COM

Complements the top number on the stack.
EXAMPLE: 200 COM . <177577> (octal radix)

DROP

Removes the top number from the stack.

SWAP

Swaps the high and low byte of the top number on the stack.
EXAMPLE: 170023 SWAP . <11760> (octal radix)

SWITCH

Switches the top two numbers on the stack.
EXAMPLE: 45 67 SWITCH . <45> . <67>

DUP

Duplicates the top number on the stack.
EXAMPLE: 12 DUP . <12> . <12>

OVER

Switches the second number on the stack with a number in the interior of the stack. The top number on the stack indicates how far down in the stack to go to get the particular number to switch with the second number on the stack.
EXAMPLE: 1 2 3 4 5 3 OVER
This switches the position of the 2 and 5 on the stack and removes the 3. Now the stack lo[illegible] like, 1 5 3 4 2. This can be seen by displaying the stack;
. <2> . <4> . <3> . <5> . <1>
Note that a "1 OVER" is the same as a "SWITCH".

UNDER

Gets a number in the interior of the stack and moves it to the top. The top number on the stack indicates how far down in the stack to go to get the particular number to be moved to the top.
EXAMPLE: 1 2 3 4 5 6 7 4 UNDER . <4>
Note that a "1 UNDER" is the same as a "DUP". A "0 UNDER" just puts zero on the stack.

AND

Performs a logical and on the top two numbers on the stack.
EXAMPLE: 7 177 AND . <7>

OR

Performs a logical or on the top two numbers on the stack.
EXAMPLE: 272 1415 OR . <1677> (octal radix)

XOR

Performs a logical exclusive or on the top two numbers on the stack.
EXAMPLE: 241 16734 XOR . <16575>

=

Tests top 2 numbers on the stack for equality. If the numbers are equal, a 1 is put on the stack. Otherwise, a 0 is put on the stack.
EXAMPLE: 12 12 = . <1>
EXAMPLE: 12 13 = . <0>
EXAMPLE: -146 -146 = . <1>

>

Tests to see if the second number on the stack is greater than the top number. If the condition is true, a 1 is put on the stack. If the condition is false, a 0 is put on the stack.
EXAMPLE: 13 12 > . <1>
EXAMPLE: 12 13 > . <0>

<

Tests to see if the second number on the stack is less than the top number on the stack. If the condition is true, a 1 is put on the stack.
EXAMPLE: 12 13 < . <1>
EXAMPLE: 13 12 < . <0>

>=

Test to see if the second number on the stack is greater or equal to the top number on the stack.
EXAMPLE: 144 144 >= . <1>

<=

Tests to see if the second number on the stack is less than or equal to the top number on the stack.
EXAMPLE: -763 -763 <= . <1>

/>/

Same as > except bit 15 is not considered to be a sign bit, it is treated as a normal data bit.
EXAMPLE: -13 12 />/ . <1>
EXAMPLE: 176500 170000 />/ . <1> (octal radix)

/</

Same as < except bit 15 is not considered to be a sign bit, it is treated as a normal data bit.
EXAMPLE: 156 -89 /</ . <1>

/>=/

Same as >= except bit 15 is not considered to be a sign bit, it is treated as a normal data bit.

/<=/

Same as <= except bit 15 is not considered to be a sign bit, it is treated as a normal data bit.

MAX

Compares the top two numbers on the stack and removes the smaller number. The larger number is left on top of the stack.
EXAMPLE: 450 23 MAX . <450>

MIN

Compares the top two numbers on the stack and removes the larger of the two numbers. The smaller number is left on top of the stack.
EXAMPLE: 23 450 MIN . <23>

/MAX/

Same as MAX except bit 15 is not considered to be a sign bit, it is treated as a normal data bit.
EXAMPLE: -453 890 /MAX/ . <-453>

/MIN/

Same as MIN except bit 15 is not considered to be a sign bit, it is treated as a normal data bit.

!

Deposits the second number on the stack at an address which is the top number on the stack.
EXAMPLE: 23 100000 ! (octal radix)
This deposits the number 23 at the address 100000. Since the ! moves an entire word, the address must be an even number so that it is on a word boundry (The ! entry will work at odd boundries with some CPU's but is is definitely bad practice).
WARNING!! Don't change the contents of memory unless you know that CONVERS is not using that memory. You can determine where the end of CONVERS is by typing DP. This will put the highest memory address CONVERS is using on the stack
EXAMPLE: o 101 o 177566 ! <A>
This deposits the ASCII code for the letter "A" at the address of the console terminal output register. This causes the letter "A" to be typed on the terminal.

@

Treats the top number on the stack as an address. The address is removed from the stack and the contents of the address is put on the stack.
EXAMPLE: 100000 @ . <23> (octal radix)
This put the contents of the address 100000 on the stack. The contents was 23. Since the @ moves an entire word, the address must be an even number so that it is on a word boundry.

BYTE!

Same as ! except only a single byte is moved. The byte to be deposited into memory should be the low byte of the top number on the stack. The byte can be deposited into the low or high byte of a memory address; therefore, the specified address does NOT need to be an even number.

BYTE@

Same as @ except only a single byte is moved. The specified address can be the low or high byte of a memory word. The byte will be moved to the low byte of the top number on the stack. The high byte of the top number on the stack will be zero.

Up till now, all examples have involved executing entries. No new entries have been created (except for HOWDY-MESSAGE). At this point, it is appropriate to start compiling entries which perform logical and arithmetic functions. When a programmer defines a new entry it becomes a part of the "user dictionary". To indicate to CONVERS that an entry is going to be defined, a colon is typed. Next, a name for the new entry must be entered. Next, the entries which will be compiled into the new entry are typed, and finally, a semicolon is typed. For instance, suppose an entry is to be defined that will multiply the top number on the stack by 13 and add 7 to it. This entry would be created by typing;

```
: TESTO 13 * 7 + ;
```

If TESTO is executed before a number is put on the stack, a stack underflow error will result. At this point, create the entry TESTO by typing the line shown above. Now, try out the entry by entering some numbers on the stack and executing TESTO;

```
DECIMAL 10 TESTO .    <137>

0 TESTO .    <7>
```

Now, the entry TESTO can be included as a part of yet another user defined entry. Try entering this entry;

```
: TEST1 MAX TESTO . ;
```

The entry TEST1 must have two numbers on the stack before it is executed. When TEST1 is executed, the larger of the top two numbers is selected (the smaller number is dropped), the number is multiplied by 13, then 7 is added. Finally, the result is output. Try this by executing;

```
10 12 TEST1    <163>
```

At this point, try to write and execute an entry that requires three numbers to be on the stack. The entry should find the smallest of the three numbers, multiply the number by 27, and output the number to the terminal.

****

SECTION IV -- CONSTANTS, VARIABLES AND ARRAYS

****

For most programming applications, it is essential that constants, variables, and arrays be used to store data, parameters, and intermediate results of calculations. It is also necessary that the stored values be accessible by any entry in the dictionary. In CONVERS, all constants, variables, and arrays are created with dictionary name headers; therefore, they are really dictionary entries. Because numbers are stored as dictionary entries, the stored values are inherently global, that is any other entry can access the stored number.

CONSTANT

Assign a name to a number.
EXAMPLE: 14 CONSTANT FOURTEEN
when FOURTEEN is executed, the number 14 will be put on the stack. Once a constant is defined, it cannot be changed.
EXAMPLE: FOURTEEN . <14>

VARIABLE

Reserves an memory location to store a variable in.
EXAMPLE: 7 VARIABLE LOOPVAR
This string creates a variable called LOOPVAR. The value of he variable is iniially 7. When LOOPVAR is executed, a memory address is placed on the stack. The contents of this address is 7. Executing LOOPVAR @ first gets the address of the variable, then gets the contents of the address, so 7 is left on the stack. To alter the contents of a variable, the desired value is deposited at the address of the variable. For example, executing 6 LOOPVAR ! will change the value of LOOPVAR to 6.
EXAMPLE: 456 LOOPVAR ! LOOPVAR @ . <456>

ARRAY

Reserves a section of memory for number storage. The top number on the stack is the number of memory locations to be reserved for the array.
EXAMPLE: 100 ARRAY DATA-ARRAY
An array called DATA-ARRAY is created which can store 100 data points. To get the address of a particular data point in the array, a displacement is put on the stack, and the array name executed. For example, to get the address of the first data point in the array DATA-ARRAY;
0 DATA-ARRAY
is executed. See this is so by outputting the address with a ".". To get the contents of this address, 0 DATA-ARRAY @ is executed. To store the number "X" in the first data point, X 0 DATA-ARRAY ! is executed. Since there are 100 data points in this array, the maximum displacement is 99.

```
        EXAMPLE: 34 10 DATA-ARRAY !
        EXAMPLE: 10 DATA-ARRAY @ .   <34>
```

To familiarize yourself with the number storage entries, try these additional examples. Remember that earlier the entry TESTO was defined which multiplied the top number on the stack by 13, then added 7. Let's try writing this entry again, but instead of using the numbers 13 and 7, use constants that are equal to 13 and 7;

```
        13 CONSTANT THIRTEEN  7 CONSTANT SEVEN

        : TESTA THIRTEEN * SEVEN + ;
```

Now try executing TESTA and confirm that it performs the same function as the entry TESTO. The VARIABLE entry is by far the most commonly used number storage entry used in programming. It allows CONVERS to be very flexible in computations etc. To demonstrate this, let's define an entry which will multiply the top number on the stack by a variable, then add the number 7. First, define the variable;

```
        0 VARIABLE *VAR
```

Now write the entry to perform the mathematical fuction;

```
        : TESTB *VAR @ * SEVEN + ;
```

If TESTB is now executed;

```
        45 TESTB .   <7>
```

The result will be 7 since the contents of the variable *VAR is zero. At this point, change the value of the variable to 13;

```
        13 *VAR !
```

and execute TESTB;

```
        45 TESTB .  <592>
```

Array storage is very similar to variable storage, the only real difference is that an offset must be specified to access a data point. Arrays are normally accessed through DO LOOPS which are described in the next section. Further examples of arrays will be postponed until DO LOOPS are described.

At this point, the student should try creating some entries which use variables and constants. Try writing an entry that will add a constant to a variable, then multiply this intermediate result by a another variable.

Before continuing to the next section, let's consider what will happen if we try to create an entry with the name TESTO. Remember that TESTO was created earlier, and unless CONVERS was exited and restarted, it still exists in the user dictionary. Try typing;

```
        : TESTO
```

CONVERS won't let you define another entry with the same name so it will reply;

```
        <TESTO IS DEFINED IN THE USER DICTIONARY>
        <CANNOT CREATE NEW ENTRY WITH THIS NAME>
```

There are several other types of error checks made when compiling a new entry. One of the most common errors is to try to compile an undefined entry, usually because of a typing error. See what happens when you do this by typing;

```
        : TESTX 1 1 + TSTO
```

CONVERS will reply with;

```
        <TSTO NO SUCH ENTRY>
```

This is just a warning message that indicates a typing error was made. You can go ahead and finish the compilation by typing the correct name;

```
        TESTO ;
```

Another warning message that may appear in a compilation is;

```
        <NUMBER TOO LARGE>
```

This indicates a number has been typed which is too large to be converted. This often happens when CONVERS is in decimal radix and you try to type in an octal number. This is also just a warning message and you can type in the correct number and continue. A different type of error occurs if the following is typed;

```
        : TESTY 1 1 + LOOP
```

CONVERS will type;

```
        <DO LOOP COMPILATION ERROR>
        <--RECOVER-->
```

The LOOP entry is an IMCD entry that will be introduced shortly. For now, all that needs to be said is that this compilation is not correct. Notice the message --RECOVER--. Most error messages in CONVERS output this to indicate a fatal error has occurred and the error was corrected through the RECOVER entry. If you are in the middle of compiling a new entry when a fatal error occurs, everything you have typed in so far has been destroyed and you must start over. In addition, any numbers that were on the stack are removed. This must be done in order to prevent compiling a entry which would crash CONVERS if executed.

****

SECTION V -- CONVERS BRANCHING AND LOOPING ENTRIES

****

CONVERS has many of the branching and looping constructs found in high level languages. The DO LOOP construct is familiar to nearly everyone who has programmed in a high level language. The unconditional branching construct, BEGIN AGAIN is not as familiar since it is designed primarily for control and data acquisition applications. The conditional branching entries, IF ELSE THEN and END sound familiar since the same names are used in PASCAL etc., but notice that in CONVERS the names have different functions which make them much more flexible and powerful.

IF
ELSE
THEN
END

These three entries make up the "IF-ELSE-THEN" conditional branch. The entry "IF" tests the top number on the stack. If the top number is not zero, any entry compiled after "IF" and any entry compiled after "THEN" is executed. (i.e. any non-zero number on the stack is treated as a "true" condition). If the top number on the stack is zero, any entry compiled after "ELSE" and any entry compiled after "THEN" is executed. The syntax for the construct is;

"IF" (entries) "ELSE" (entries) "THEN" (entries)

The "ELSE" is optional, but "IF" and "THEN" must be present. Entries compiled after "IF", "ELSE", and "THEN" are also optional. The END entry is a special entry which causes unconditional termination of the entry in which the "IF-ELSE-THEN" construct is compiled. When END is executed, entries compiled after a "THEN" will NOT be executed.

EXAMPLE: <u>: TST LOOPVAR @ 10 > IF BELL ELSE CRLF THEN SPACE ;</u>

The entry TST will check the contents of the variable LOOPVAR (LOOPVAR was defined earlier, remember?) If the contents of LOOPVAR is greater than 10, a BELL will be executed, followed by a SPACE. If the contents of LOOPVAR is less than or equal to 10, a CRLF will be executed followed by a SPACE. Try this by depositing numbers greater than and less than 10 in LOOPVAR and executing TST.

EXAMPLE: <u>: TST0 LOOPVAR @ 10 >= IF BELL THEN ;</u>

In this example, if the contents of LOOPVAR is greater than or equal to 10, a BELL will be executed. If the contents of LOOPVAR is less than 10, nothing will be executed. In this example, "ELSE" is not required.

EXAMPLE: <u>: TST1 LOOPVAR @ 776 = IF BELL ELSE THEN ;</u>

In this example, if the contents of LOOPVAR is equal to 776, a BELL will be executed, otherwise nothing will be executed.

EXAMPLE: <u>: TST2 LOOPVAR @ IF END THEN BELL ;</u>

In this example, if the contents of LOOPVAR is zero, a BELL will be executed, otherwise nothing will be executed. Try all of these examples by changing LOOPVAR and checking the results when the new entries are executed.

BEGIN
AGAIN

These entries form one type of looping construct commonly used in CONVERS. A BEGIN AGAIN loop construct is used when the number of loops to be made is infinite or undetermined. The BEGIN entry marks the start of a loop, and the AGAIN entry marks the end of the loop. When the number of loops to be made is undetermined, the BEGIN AGAIN loop construct is most often used in conjuction with the conditional branching entries IF, ELSE, and THEN.
EXAMPLE: : TST3 BEGIN BELL AGAIN ;
This is an infinite loop which rings the terminal bell. This loop can only be broken with operator intervention (hitting a ©C). The BEGIN marks the beginning of the loop, and AGAIN causes a jump back to BEGIN.
EXAMPLE: : TST4 BEGIN CSR BYTE@ IF ELSE AGAIN THEN ;
In this example, the processer remains in a tight loop as long as the contents of CSR is zero. When the contents of CSR is nonzero, the loop is broken. This is how a "ready" bit is checked on a peripheral device for input/output applications. TST4 cannot be entered at this point because CSR has not been defined. For I/O applications, CSR would be a control and status register on an interface board.

DO
I
J
IJ
K
LOOP

These entries form the most common looping structure used in CONVERS because the programmer specifies the number of loops to be executed. The entry DO marks the beginning of the loop, and the entry LOOP marks the end of the loop. The I,J,K, and IJ entries put loop indexes on the stack. Do loops can be nested but an inner loop must be
completely enclosed within an outer loop. The I entry gets the loop index of the innermost loop and puts it on the stack. The entries J and K get the loop indexes of successive outer loops. The IJ entry is special and will be described by example.
EXAMPLE: : TST5 10 0 DO BELL LOOP ;
The entry TST5 rings the bell on the terminal 10 times. The number 10 preceding DO specifies the loop should execute 10 times. The 0 specifes that the loop index (I) should start at zero and count up. Notice that this convention is very different from most other languages!
EXAMPLE: : TST6 10 0 DO I . LOOP ;

The "I" entry put the loop index on the stack each time through through the loop. Because the index was specified to start at zero, this example outputs the numbers 0 through 9 to the terminal.

EXAMPLE: : TST7 10 3 DO I . LOOP ;

This entry outputs the numbers 3 through 12 to the terminal.

EXAMPLE: : TST8 3 0 DO 10 0 DO J . I . LOOP CRLF LOOP ;

This is a two dimensional loop. The "J" puts the loop index of the outer loop on the stack and the "I" puts the loop index of the inner loop on the stack. In this example, he numbers;

```
0 0 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8 0 9
1 0 1 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
2 0 2 1 2 2 2 3 2 4 2 5 2 6 2 7 2 8 2 9
```

are output on the terminal.

EXAMPLE: : TST9 3 0 DO 10 0 DO IJ . LOOP CRLF LOOP ;

The entry TST9 will output;

```
0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
```

to the terminal. Notice that the IJ entry is equal to J (outer loop count) 10 * I +. The IJ entry was created specifically for accessing sequential data points in an array from nested DO LOOP's.

DO LOOP's are used only when then number of "loops" to be made is known exactly. If the number of "loops" to be made is not known, the BEGIN-AGAIN construct must be used. An IF-ELSE-THEN construct can NOT be used to branch out of a DO-LOOP. If this is attempted, CONVERS will crash.

The branching and looping entries are by far the most useful entries in CONVERS. Without these entries CONVERS would strictly be a "low level" language. With these entries, CONVERS is able to span the range of high to low level in one language. DO LOOPS are very useful for accessing data points in arrays, as was mentioned earlier. As an example of DO LOOPS and arrays, suppose data is to be put into an array, then each data point is to be divided by 5.

Remember a 100 point array was created earlier called DATA-ARRAY. Now create an entry that will initialize all points in the array to zero;

```
: CLEAR 100 0 DO 0 I DATA-ARRAY ! LOOP ;
```

and another entry to display the array on the terminal. One hundred points will not fit on the terminal on one line, so the display entry should output 10 lines of 10 data points per line;

```
: DISPLAY 10 0 DO 10 0 DO IJ DATA-ARRAY @ . LOOP CRLF LOOP ;
```

Now, make sure these entries work by executing them;

```
CLEAR DISPLAY
```

A pattern of 100 zeros should appear on the terminal. Now, write an entry that will add a variable to each point in the array;

```
        0 VARIABLE +VAR

        : ADD3 100 0 DO +VAR @ I DATA-ARRAY @ + I DATA-ARRAY ! LOOP ;
```

Try out these entries by executing;

```
        568 +VAR ! ADD3 DISPLAY
```

The number 568 will appear on the terminal 100 times.

At this point, try to write entries based on DO LOOPS that will put the numbers 0 through 99 in sequential data points of the DATA-ARRAY. Point 0 should contain 0, point 9 should contain 9, and point 99 should contain 99. Next , try writing an entry that will display the contents of DATA-ARRAY as 4 columns and 25 rows. Just to keep things interesting, ring the terminal bell every time a new row is output. The branching and looping entries are the most flexible, and therefore the most complex entries in CONVERS. Make sure you have tried all of the examples and fully understand how each of the branching and looping constructs work before continuing.

****

SECTION VI -- OTHER CONVERS ENTRIES

****

As you might guess, there aren't many more different classes of CONVERS entries to discuss. The following are assorted entries that can't be classified with the type of entries discussed so far.

EXIT

This causes CONVERS to stop running and reboot the RT-11 operating system.

FORGET

The FORGET entry removes entries from the user dictionary.
EXAMPLE: FORGET TST3
The entry TST3, and all entries defined after TST3 are removed from the dictionary. Try this and verify that it works by trying to executing TST3 after it is "forgotten".

FIND

Puts the starting address of a CONVERS dictionary entry on the stack. FIND is a high precedence entry so it cannot be compiled.
EXAMPLE FIND TST2
The top number on the stack will now be the starting address of the TST2 entry. If the entry is not found in the dictionary, the message;
NO SUCH ENTRY
is output to the terminal. Try this by trying to "find" the entry TST3.

****
SECTION VII -- DISK INPUT AND OUTPUT IN CONVERS
****

The most common disk operating system used with the smaller PDP-11 and LSI-11 computers is RT-11. This is a very well refined operating system since it has been in existence for a number of years. Because so many people are familiar with RT-11 and so many programs are written to run under it, the file structure in CONVERS emulates the RT-11 file structure. Since RT-11 has all of the utilities needed to perform such functions as disk formatting and initialization, directory maintenance, etc., these utilities are not included in CONVERS. The main advantage in having CONVERS disk files compatible with the RT-11 files is that information can be passed between CONVERS and other programming languages. For instance, CONVERS can be used to collect data from an experiment, then write the data to a disk file. A program written in FORTRAN can then be used to analyze the data. When file names are specified in CONVERS, it must be compatible with RT-11. A name can consist of the letters A through Z and the numerals 0 through 9. Up to six characters specifiy the file name, and three chracters the file extension. The name and extension are separated by a period.

LOAD

Moves a CONVERS source file from disk to memory and executes and/or compiles the source as if the input were being typed from the terminal. One limitation in using the LOAD entry is that the file being loaded cannot load another file simultaneously. Once the LOAD is finished, the last entry compiled in the dictionary will be executed, irregardless of what it is. This provides the facility to have a program start itself once it is loaded. If you do not want to use this feature, make the last entry in the source file a dummy entry, i.e., one that does nothing, such as;
: DUMMY ;
The default file extension for a CONVERS source file is .CON.
EXAMPLE: <u>LOAD FP.CON</u>
EXAMPLE: <u>LOAD LP</u>
Trying to load these files will result in the message;
<FILE NOT FOUND>
<--RECOVER-->
If the files are not on the disk.
At this time, CONVERS does not have a text editor. To write a CONVERS source file, use any desired editor which operates under RT-11, such as TECO. When writing a source file, enclose comments in parentheses such as;
: GO GET-DATA VARIANCE ; (get data, calculate variance)
Anything in parentheses will be totally ignored.

WRITE-IMAGE

Writes the entire user dictionary to the disk under an assigned file name. This type of file is called a memory image file and

it contains compiled CONVERS source code. Reading and writing memory image files eliminates the compilation step required when a source file is loaded. The default file name extension is .IMG
EXAMPLE: WRITE-IMAGE DICT.IMG

GET

Reads a memory image file from disk to the dictionary from a named file. A memory image file is created by compiling source code from the terminal or disk, (using the LOAD entry), then writing the memory image to disk (using the WRITE-IMAGE entry). The default file name extension is .IMG. Just as in the LOAD entry, once the GET is completed, the last entry in the dictionary is executed. Getting a memory image file is much faster than loading a source file because the compilation step is avoided. A memory image file should only contain entries which have been loaded from a source file, not typed from the terminal. Also, the source file should always be retained because an image file can never be "de-compiled". Before executing the next example, "forget" the first entry in the user dictionary, MESSAGE. Make sure it's gone by trying to "find" MESSAGE, then execute;
EXAMPLE: GET DICT
And confirm that all entries are back in the user dictionary.

WRITE

Writes data from a named array to a named disk file. If a disk file is found with the same name as the specified file, the old file is deleted before the new one is written. The default file name extension is .DAT. To remember the syntax for the WRITE entry, think write from array, to data file.
EXAMPLE: WRITE DATA-ARRAY DATA1

READ

Reads data from a named disk file to a named array. The array must have been defined previous to the read and it must be large enough to hold all of the data. The default file name extension is .DAT. To remember the syntax for the READ entry, think read from data file to array.
EXAMPLE: READ DATA1 DATA-ARRAY
Try writing the DATA-ARRAY, change the DATA-ARRAY, then read back from disk and verify that DATA-ARRAY is restored.

DELETE

Deletes a named file. A file extension must be specified as there are no defaults.
EXAMPLE: DELETE DATA1.DAT

[ ]

This forces CONVERS into an interpreter mode while executing a compiled entry. This is the only way file names may be specified as part of a user defined entry. At compilation time, the ASCII characters of everything enclosed within brackets is put into memory directly. When the entry is executed, the input routine fetches the ASCII characters from memory, just as if they were being typed from the terminal. The right and left brackets are used to enclose the entry to be executed and the disk file and data array which the disk entry is to operate on. Only one command may be enclosed within a single set of brackets.

EXAMPLE: <u>: DATA-WRITE [WRITE DATA-ARRAY DATA1] ;</u>

Now try executing this entry to verify that it works. Note that this entry;

```
: LOAD+WRITE [ LOAD FP WRITE DATA DATA1 ] ;
```

is not legal because two commands are enclosed in one set of brackets. LOAD+WRITE should be written as;

```
: LOAD+WRITE [ LOAD FP ] [ WRITE DATA DATA1 ] ;
```

CHOOSING THE DISK UNIT TO READ OR WRITE TO

The portion of CONVERS which directly interacts with the disk is called the resident disk handler. The number and type of resident disk handlers in CONVERS is dependent on the disk hardware available for a particular computer system. When CONVERS is first started, a message is typed on the terminal which indicates which resident handlers are included in the IMCD for that particular system. Also, the disk unit and drive which CONVERS will initially communicate with when first started is dependent upon how CONVERS was generated (see appendix A for details on generating CONVERS). All resident disk handlers have within them, entries which select the disk unit and drive to which data will be read and written. The names of these entries correspond to the RT-11 names for a given unit and drive. Presently, resident handlers exist for the RX01 single density diskette unit, the RX02 double density diskette unit, and the RL01 and/or RL02 hard disk. The entries to select a disk unit and drive are then;

for the RX01;	DX0 (drive 0) and DX1 (drive 1)
for the RX02;	DY0 (drive 0) and DY1 (drive 1)
for the RL01/RL02;	DL0 (drive 0) and DL1 (drive 1)

Notice that unlike RT-11, there is no colon following the name of a drive. In CONVERS, when a particular drive is selected for reading or writing, all disk transfers will occur with that drive until a different drive is selected, there is no "system disk" as there is in RT-11.

As an example, suppose it is desired to copy a data file from drive 0 to drive 1 of a RX02 diskette unit. First, the file must be read from drive 0 into an array;

```
DY0 READ DATA1 DATA-ARRAY
```

and now the array written to drive 1;

```
        DY1 WRITE DATA-ARRAY DATA1
```

At this point, if another disk transfer is executed, such as;

```
        LOAD FP
```

CONVERS will go to drive 1 to find FP.CON, it will not go to drive 0 as RT-11 would. If there is no disk in drive 1, the message;

```
        RX02 DISK ERROR
        --RESET--
        --RECOVER--
```

will appear.

ACKNOWLEDGMENT

This work was supported in part by the Office of Naval Research.

REFERENCES

1. C. Moore, Astronomical Astrophysics Supp. 15, 497 (1974).

2. S.B. Tilden and M.B. Denton, J. Auto. Chem. 1, 128 (1979).

3. Scott B. Tilden and M. Bonner Denton, CONVERS An Interpretive Compiler, Department of Chemistry, University of Arizona, Tucson, Arizona 85721 (1980).

4. Arthur Gill, Machine and Assembly Language Programming of the PDP-11, Prentice-Hall, New Jersey (1978).

5. James W. Cooper, The Minicomputer in the Laboratory, Wiley, New York (1983).