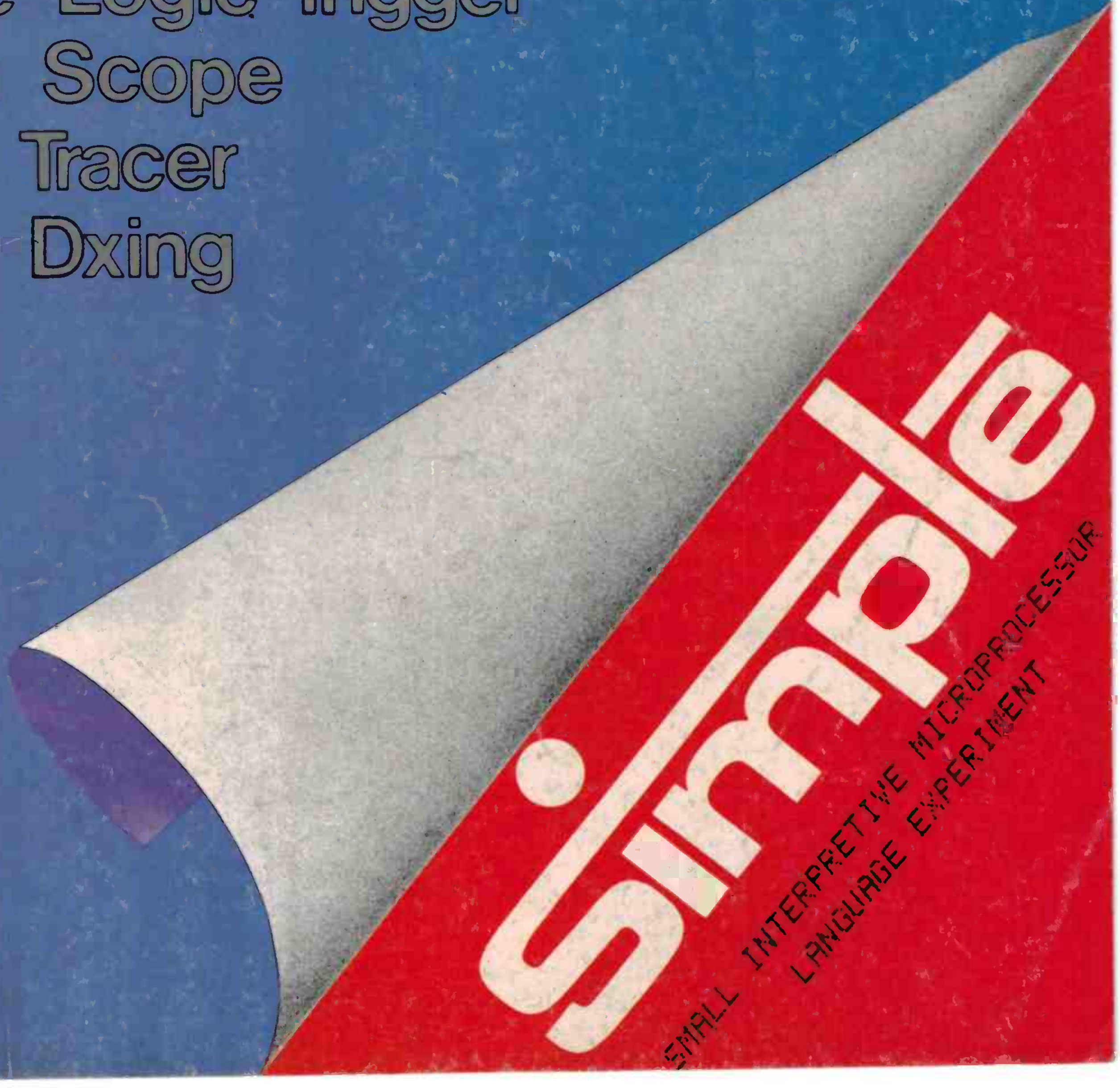
electronics today INTERNATIONAL

JANUARY 1979

$1·40*
NZ $1.50

2m VMOS Power Amp
Race Track Game
Scope Logic Trigger
Use a Scope
Curve Tracer
Utility Dxing



Simple

A small interpreter for microprocessors, designed by Dr. Tim Hendtlass of RMIT.

PROGRAMMING WITH A high level language interpreter is much easier than programming in absolute machine code, but most interpreters themselves occupy several thousand bytes of RAM. Yet more memory is needed for the user program written in the high level language. This usually restricts people whose microcomputers have only a limited amount of memory to machine code programming.

To help such people explore the world of interpreters, over the past few months I have written a small one for the 8080/8085/Z80 family of microprocessors. It has taken this long because it has been mainly done in short bursts, much of the work being carried out on trams in order to make profitable use of what is otherwise wasted commuting time. Early attempts were based on minimal implementations of BASIC, and were not very successful. While it is possible to fit an integer-only small BASIC into about 1000 bytes the very nature of BASIC does not lend itself to much more shrinking without becoming of very little use at all. Rethinking the problem I decided that the language I wanted had to be small enough to be able to be entered manually, if necessary, whenever you wanted to use it. Also quite reasonable user programs should be able to be run in a pittance of memory. I also decided that it should be coded in a reasonably straightforward way so it should permit users to try out their own additions without having to make any major changes to my code. I gave myself a budget of 256 bytes for the whole interpreter including its text editor and set about seeing how much could be achieved in that. The answer may well surprise you.

I have taken much inspiration from the CAI language PILOT which really lends itself to compact versions. Naturally I have had to omit quite a bit that the full PILOT offers, but I have added a few things too.

Enough background. Here then is a *finished product (it works),* a not too difficult exercise in reading a machine language program (see how it works) and a project for you to add to and modify as much as you like (make it work your way). Since this is not PILOT I have had to find another name for it. As this has been my Small Interpretive MicroProcessor Language Experiment, I have called my language SIMPLE.

Just what is SIMPLE?

If one is not careful confusion will arise because there are actually two things both of which are named SIMPLE. First, there is the actual SIMPLE interpreter which is an 8080 machine language program. This interpreter receives the instructions that actually make it do something in a special language also called SIMPLE. It is this SIMPLE language which will be described in this first part of the article and examples given of its use. A source listing of the actual interpreter and a description of what makes it tick is the subject of the second part, along with some hints on implementing the SIMPLE interpreter on other microprocessors.

For ease of reading for the rest of this first part, Simple means the SIMPLE language unless otherwise stated.

The SIMPLE Language

Simple consists of two basic types of things, commands and statements. Commands are things that are to be done at once while statements are things to be stored away to be done (executed) at some later time. A sequence of statements is called the user's program.

Simple has only five commands. One of these causes the interpreter to start executing the user's program that has been previously stored away. The other four are all to do with entering the user's program and correcting it if it does not do what you want. It is probably better to first consider what statements are available to you, the programmer, and come back later to see just how to get these statements stored in memory.

Simple is a line orientated language, that is the stored program is divided into a series of lines. A line is any group of characters between two carriage returns. Starting at the beginning of the program the first line consists of all the characters up to and including the first carriage return. The second line is all the characters after this carriage return up to and including the next carriage return and so on. There can be one or more statements on a line and statements may themselves consist of one or more characters. The first letter of every statement is called the key letter and serves to identify the kind of statement. Now let's meet some statements.

Six Fundamental Statements

The Type Statement (T)

This must start with a T and then may be followed by any collection of characters you like except *, %, #, $, <— and &. It must end with a carriage return. The effect of this statement when encountered as part of a user's program is to print everything after the key letter T up to the carriage return at the end of the statement.

For example (in the following a carriage return is indicated by (CR)),

T THIS IS THE FIRST LINE (CR)
T THIS IS THE SECOND. (CR)

would produce

THIS IS THE FIRST LINE
THIS IS THE SECOND.

The Accept Statement (A)

This is complete with just an A. It causes a single character to be accepted from the operator and stored internally in the last input character buffer.

The Match Statement (M)

The key letter M must be directly followed by another character. This character is compared with the last character input in response to an ACCEPT statement. If they are the same an internal flag is set to "YES". if they are not the flag is set to "NO".

The Yes Statement (Y)

This is complete with just a Y. When encountered the state of the YES/NO flag is inspected. If it is set to "YES" the statement directly after the Y is executed, but if it is set to "NO" all the rest of the statements after the Y on this line are passed over and the next statement executed is the first statement on the following line.

The No Statement (N)

This is like the Y statement above, except that now only if the flag is set to "NO" will the statement directly after the N be executed, otherwise execution continues with the next line.

The End Statement (E)

This is complete in itself and causes execution of the user's program to be stopped and the Simple Interpreter to sit and wait for a command. When an E statement is found an E is always typed automatically to tell the operator that the end of the user's program has been reached. (Note that the user program pointer (of which more later) is reset to the beginning of the user program when an end statement is executed.)

To illustrate all of the statements so far consider the following (each line finishes with a carriage return, but these are not shown for ease of reading):—

```
T     PLEASE TYPE ME  A  Q
A
MQ
YT – THANK YOU
NT – IS NOT A  Q!
E
```

The first line causes a message to be typed requesting the operator to reply with a 'Q'. The second line accepts a character and the third matches it to see if it is a 'Q'. If it is, the 'yes' flag is set and so the 'thank you' message is typed. Since the 'yes' and 'no' flags cannot be set at once, in this case the program skips the next line and ends. If it was not a 'Q' that was entered the match statement would have given a 'No' answer and so the complaint would have been typed and then the program would end at the E statement. The actual output from the program would be (input from the operator is underlined here just to identify it):—

Case 1.	Case 2.
PLEASE TYPE ME A Q	PLEASE TYPE ME A Q
Q – THANK YOU	R – IS NOT A Q!
E	E

More Statements — Jumps and Subroutines

The example above is reasonably trivial deliberately to make it easy to follow and far more complex things can be done with just the six statements introduced so far. However, the power of the language is increased enormously as soon as jumps are introduced. Both backward and forward jumps and subroutine calls are allowed. Subroutines may not be nested, in other words a subroutine may not call another subroutine or itself. It did not seem worthwhile to permit more than one level of subroutine, but if you disagree you can reasonably easily modify the machine code interpreter to permit this. I doubt it will still fit into 256 bytes though.

Both jumping and subroutining require some way to identify the destination and the special character * is reserved for this purpose. This is known as a marker and the first one starting from the beginning of the program is marker one. The next one is marker two and so on. It is optional, if you wish, to precede an * by its number when entering the program. This can help you to understand the program flow. Thus you could enter 3* for the third marker and 4* for the fourth, or you could delete the 3 and 4. It makes no difference to user program execution. Up to nine markers may be used in a program.

The Jump Statement (J)

The marker number must directly follow the J, for example J3. This would transfer control to the statement directly following the third occurrence of an * in the user's program. This can be anywhere at all, at the beginning, middle or end of line.

The Subroutine Statement (S)

Exactly as J above except that the address of the statement after the subroutine call is saved before control is passed to the statement after the target *.

The Return Statement (R)

Complete in itself as just an R. Used at the end of a subroutine to return to the statement after the S statement that brought us to this subroutine.

More things you may do

In Simple it is easy and usually desirable to put more than one statement on a line. No special character is required to separate the statements, but it is advisable to use one to improve the readability of the program. Because of the way Simple has been written any character that comes before A in the ASCII character set and which has not already been allocated some other special function may be used. Thus you could use a space, a comma or a colon to name but three. See Table 1 for the full range of printing characters open to you to use as statement separators. Any number or combination of these may be used between statements to improve program readability. One restriction on using multiple statements on a line; a type (T) statement must be the last one to appear on its line.

To further help improve program readability a comment statement has been introduced. This consists of a leading C followed by anything at all. The line is *always* skipped over on program execution, but will be printed when you list the program. There is no point putting any further statements on a line after a comment statement as Simple will never see them.

Putting more than one statement on a line is more than a convenience; it also permits more complex checks than just the single character match. Suppose you wish to check if the operator has given one of a number of possible answers. This can be done by using the fact that a Y or N comment, if not met, causes the whole of the rest of the line to be skipped. See program 1 which uses this in practice in the third line. In all the program examples shown operator input has been underlined in the sample run for clarity. The & at the beginning of each line of the listing is the command which causes that line to be printed (more of the commands later). If an A has been entered it is matched and the first No test is failed and the program skips to line four arriving with the Yes flag set. If it was not an A a match to E is tried, only if

```
& 1*  T TYPE ME A LETTER
&     A
&     MA,NME,NMI,NMO,NMU
&     YT IS A VOWEL
&     NT IS NOT A VOWEL
&     T DO YOU WANT TO TRY AGAIN? IF SO TYPE 'Y'
&     A,T
&     MY,YJ1
&     E

1 TYPE ME A LETTER
S IS NOT A VOWEL
 DO YOU WANT TO TRY AGAIN? IF SO TYFE 'Y'
Y
 TYPE ME A LETTER
U IS A VOWEL
 DO YOU WANT TO TRY AGAIN? IF SO TYFE 'Y'
N
E
```

Program 1 (left). Vowel test.

```
&1* T PLEASE ENTER A LETTER EXCEPT A OR Z
&   A,T
&   MA,NMZ
&   YJ4
&   S2
&   P,T IS THE NEXT LETTER
&   S3
&   P,T IS THE PREVIOUS LETTER
&   J1
&2* X,I,X,R
&3* X,D,D,X,R
&4* T I SAID NOT A OR Z
&   J1

R$ PLEASE ENTER A LETTER EXCEPT A OR Z
D
E IS THE NEXT LETTER
C IS THE PREVIOUS LETTER
 PLEASE ENTER A LETTER EXCEPT A OR Z
Z
 I SAID NOT A OR Z
 PLEASE ENTER A LETTER EXCEPT A OR Z
```

Program 2 (lower left). Alphabet program.

```
&    T PLEASE SPELL THE WORD FOR 2 FOR ME
&    T (HINT IT HAS 3 LETTERS)
&1*  A,NT,A,YMW
&    A,YMO,YT IS CORRECT
&    YE
&    T IS NOT RIGHT,SORRY. IF YOU WANT TO
&    T TRY AGAIN PRESS 'Y'
&    A,T
&    MY,YJ1
&    E

R$ PLEASE SPELL THE WORD FOR 2 FOR ME
 (HINT IT HAS 3 LETTERS)
TOO IS NOT RIGHT,SORRY. IF YOU WANT TO
 TRY AGAIN PRESS 'Y'
Y
TWO IS CORRECT
E
```

Program 3 Spelling program.

that fails is a match to I tried and so on. At line four the yes flag is set if the character entered is any one of the vowels, A, E, I, O or U. This program shows a conditioned jump being used back to the start of the program in line eight.

It is also easy to test multi-letter answers. See program 2. Here the first letter input is matched in line three to T. If this match fails no further testing is done although two other letters are accepted. Only if the correct three letters are entered in the correct order will the yes flag be set at line eight.

For an example of a program which uses subroutines (although it could be written without them) see program 3. This program forms an infinite loop; once started the program can only be stopped by resetting the computer — not usually desirable.

Yet More Statements

The statements introduced so far do not permit you to do anything more with the single character response to an accept statement than match it against one or more approved answers. Simple also lets you save (keep) and recall (get) characters for later use. There are nine storage locations set aside for you to use as single character memories and these are identified by the characters 1 to 9 as explained below.

The Keep Statement (K)

The memory number must directly follow the K with no other character in between, e.g. K3. The contents of the last character entered buffer are copied into memory 3 replacing what was there. The contents of the last character entered buffer are unaltered.

The Get Statement (G)

Again, the memory number must directly follow the key letter G, e.g. G3. The contents of memory 3 replace the contents of the last character entered buffer. The contents of memory location 3 are not altered.

When a character is entered in response to an accept statement it is automatically printed on the terminal as it is put in the last character entered buffer. Now that we can keep and get previously entered characters it is desirable to be able to print whatever character might be stored in the last character buffer at any time. This is done with the P command.

The Print Command (P)

Complete in itself as just P. Causes the current contents of the last character entered buffer to be printed on the terminal.

Even More Statements Still!

As you will have noticed, so far Simple is a mathematical simpleton. There is one group of statements which let you do some very limited maths; sufficient for counting tries or playing some games. Simple has a built-in counter which you may load (L), increment (I) or decrement (D). You also need to be able to inspect the counter contents at any time to see if they have reached some special value. This is achieved by providing an exchange (X) statement, a statement which opens far more scope than might at first appear. X swaps the counter contents for the contents of the last character input buffer — note this is a swap not a copy, so no information is destroyed. Once X has been issued the counter contents may be matched to some special value (using M), kept (using K) and/or printed (using P). The counter and last character input can be replaced in their normal places by a second X statement.

Also the X statement, used with the I or D statements, permits a user response to be cycled up or down. The G and X statements together allow the current counter contents to be replaced by some value previously stored away. These features will be used considerably in the examples at the end of this part.

The Load Counter Statement (L)

The character immediately following the L is placed into the counter replacing what was there. For example, L7 loads the counter with the character 7.

The Increment Statement (I)

Complete in itself. Causes the counter contents to be increased by one.

The Decrement Statement (D)

Complete in itself. Causes the counter contents to be decreased by one.

The Exchange Statement (X)

Complete in itself. Causes the counter contents to be exchanged with the contents of the last character entered buffer.

A few more bits and pieces

Simple was also designed for you to add on to without any major changes to the interpreter source code. This is done through the U or User statement. Until you change two bytes in the source code this will be flagged as an error. More on using this U statement in the second part of this article.

In fact a number of things can be wrong with a Simple program; you might use an illegal key letter or refer to a non-existent memory register for example. If Simple finds such a problem it stops execution, prints a ?, prints the very statement which has caused the trouble, prints the rest of the current line and then reverts to the command mode, sets the entry pointer to the beginning of the user program and waits for you to do something about it. Another possible error is to try to type a line of more than 64 characters. After Simple has typed 64 characters without finding a carriage return, it prints a ? and reverts to the command code.

The Command Mode

Having now discussed the statements you have available when writing a program in Simple, it is time to describe how to get a program into the user memory space and how to start executing it.

The text editor provided as part of the total Simple interpreter is of necessity fairly limited. There are five printing characters reserved for it; anything else entered will be treated as something to be put into the user's program. Characters are put sequentially into the user's program area at the position of the entry pointer using one byte of storage per character. If you enter an incorrect character, entering back space (← or control H) will remove the last character. A number of back spaces can be used to remove a string of incorrect characters.

At any time you can return the entry pointer to the beginning of the user's program area by entering #. An & causes the next line of program in the memory starting at the current position of the entry pointer to be displayed on the terminal and the entry pointer moved to the start of the following line. You can replace a line for one of equal or shorter length by going to the start (#) and displaying (with &'s) up to the line before the one you wish to replace. Then type the new line ending with a % instead of a carriage return. Any characters left from the old line will be erased for you. This also lets you erase a superfluous line if you wish; you do not type in a new line, just enter %.

To run a program enter # to get back to the start (assuming you want to start at the beginning — you may start anywhere you wish), and then enter $ to start the program executing. Provided you end your program with an **E** command, after execution is finished control will be returned to the command mode. An error always brings you back to the command code.

Simple Examples

As mentioned before, Simple is not a mathematical tool — while you can do multi-digit arithmetic on it, it is not easy. However, you can do a wide range of things involving logic trees and/or alphanumeric character manipulations. To conclude this section I give some more examples. The examples are each in two parts; first a listing of the program itself (produced using the & command described above) and then a sample output from the program. For clarity all input from the operator is underlined. You will notice that I have used the optional features of writing a Simple program to make these listings look tidy. Remember if you wish to save memory space you can eliminate all blanks, commas and the number before each marker.

The fourth program shows a simple game of NIM. In this version the computer is unbeatable, but a more complex one can be produced which gives the operator a chance! The fifth (a more complicated program) shows an alphabetic version of HI-LO in which you have to guess the mystery letter from the 'too high' or 'too low' clues. It is almost always possible to get to the answer in five tries — here you are only given four. Note the way of selecting the 'unknown' number based on responses from the user. Unfortunately, there was no room

Program 4. NIM.

```
&   C SET UP THE PROBLEM
&1* L=,T DO YOU WANT INSTRUCTIONS? Y OR N
&   A,T
&   MY,NJ2
&   T WE START OFF WITH 13 MATCHES AND TAKE TURNS
&   T REMOVING 1,2 OR 3 MATCHES. THE PERSON TO HAVE
&   T TO TAKE THE LAST MATCH LOSES. YOU GO FIRST.
&2* T THERE ARE NOW 13 MATCHES.
&   J4
&3* X,P,X,T MATCHES LEFT
&4* T HOW MANY DO YOU TAKE?
&   A,T
&   M1,NM2,NM3
&   NT EITHER 1 OR 2 OR 3 PLEASE
&   NJ4
&   C HAVING GOT THEIR NUMBER WE WORK OUT OURS
&   M1,YT I TAKE 3
&   M2,YT I TAKE 2
&   M3,YT I TAKE 1
&   D,D,D,D,C WE WON?
&   X,M1,X,YJ5,C IF SO JUMP TO FIFTH MARKER
&   J3,C OTHERWISE BACK FOR NEXT ROUND
&5* T I WIN - WANT ANOTHER GAME? (Y OR N)
&   A,T
&   NY,YJ1
&   T OH WELL,BEEN NICE PLAYING YOU!
&   E

 #$ DO YOU WANT INSTRUCTIONS? Y OR N
 Y
  WE START OFF WITH 13 MATCHES AND TAKE TURNS
  REMOVING 1,2 OR 3 MATCHES. THE PERSON TO HAVE
  TO TAKE THE LAST MATCH LOSES. YOU GO FIRST.
  THERE ARE NOW 13 MATCHES.
  HOW MANY DO YOU TAKE?
 3
  I TAKE 1
 9 MATCHES LEFT
  HOW MANY DO YOU TAKE?
 4
  EITHER 1 OR 2 OR 3 PLEASE
  HOW MANY DO YOU TAKE?
 2
  I TAKE 2
 5 MATCHES LEFT
  HOW MANY DO YOU TAKE?
 1
  I TAKE 3
  I WIN - WANT ANOTHER GAME? (Y OR N)
 N
  OH WELL,BEEN NICE PLAYING YOU!
 E
```

Program 5. HI-LO.

```
#$ PLEASE GIVE ME 3 DIFFERENT LETTERS
ETI - THANK YOU
 YOUR GUESS?
M - IS TOO LOW
 YOUR GUESS?
S - IS TOO HIGH
 YOUR GUESS?
P - IS TOO HIGH
 YOUR GUESS?
N - IS CORRECT IN
4 TRIES!
 WANT ANOTHER GAME? (Y OR N)
Y YOUR GUESS?
E - IS TOO LOW
 YOUR GUESS?
Q - IS TOO LOW
 YOUR GUESS?
S - IS TOO LOW
 YOUR GUESS?
X - IS TOO HIGH
 YOU HAVE HAD 4 TRIES,
U WAS THE CORRECT ANSWER
 WANT ANOTHER GAME? (Y OR N)
NE

&   C GET 3 CHARACTERS TO FORM CORRECT ANSWER
&   T PLEASE GIVE ME 3 DIFFERENT LETTERS
&   A,K1,A,K1,A,K3,T - THANK YOU
&   C WORK OUT CORRECT ANSWER AND SET NO OF TRIES TO 0
&1* G1,X,G2,S7,G3,X,S7,X,K4,L0,X,K5
&   C SEE IF THEY HAVE USED UP 4 TRIES
&2* G5,X,I,X,M5,K5,NJ4
&   T YOU HAVE HAD 4 TRIES,
&   G4,P,T WAS THE CORRECT ANSWER
&3* T WANT ANOTHER GAME? (Y OR N)
&   A,MY,YJ1
&   NE
&   C GET THEIR GUESS,CHANGE R1,R2,R3 TO GET A DIFFERENT
&   C CORRECT ANSWER NEXT TIME.PUT THEIR GUESS IN COUNTER
&   C AND CORRECT ANSWER IN LAST CHARACTER INPUT BUFFER.
&4* T YOUR GUESS?
&   A,K6,G2,K1,G3,K2,G6,K3,X,G4
&   C WORK OUT OUR REPLY BY CYCLING EACH CHARACTER DOWN
&   C TO A AND SEEING WHICH GETS THERE FIRST.
&5* MA,YJ6
&   X,MA,YT - IS TOO LOW
&   YJ2
&   D,X,D,J5
&6* X,MA,NT - IS TOO HIGH
&   NJ2
&   T - IS CORRECT IN
&   G5,P,T TRIES!
&   J3
&   C SUBROUTINE TO CYCLE COUNTER UP WHILE BRINGING THE
&   C LAST CHARACTER ENTERED BUFFER DOWN TO A
&7* MA,YR
&   X,NZ,X,YL0
&   I,X,D,X,J7
```

Simple

```
                ;SMALL INTERPRETIVE MICROPROCESSOR LANGUAGE EXPERIMENT

                ;*********************** SIMPLE ***********************

                ;WRITTEN BY TIM HENDTLASS
                ;OCTOBER 1978
                ;VERSION 2.1 25/10/78


                ;FIRST TELL THE ASSEMBLER WHERE THE PROGRAM IS TO START

0000                    ORG     100H

                ;THE PROGRAM PROPER STARTS HERE

                ;THIS IS THE MASTER TEXT EDIT ROUTINE

0100 211502     START:  LXI     H,UPROG ;POINT H&L TO THE START OF THE
                                        ;USER PROGRAM AREA
0103 F9                 SPHL            ;STACK WILL START JUST BELOW THE
                                        ;USER PROGRAM
0104 110401     TLOOP:  LXI     D,TLOOP ;TLOOP ADDRESS TO D&E
0107 D5                 PUSH    D       ;AND ONTO THE TOP OF THE STACK
                                        ;THIS LETS US RETURN TO TLOOP
                                        ;WITH A RETURN INSTRUCTION
0108 CDF301             CALL    CI      ;GET (AND ECHO) A CHARACTER
010B 2B                 DCX     H       ;IN CASE IT IS BACKSPACE
010C FE5F               CPI     5FH     ;WAS IT BACKSPACE (5F)?
010E C8                 RZ              ;YES,WE HAVE ALREADY DONE THE
                                        ;NECESSARY CORRECTION,SO BACK
                                        ;TO TLOOP FOR THE NEXT CHARACTER
010F 23                 INX     H       ;IT WASN'T,SO RESTORE H&L AS
                                        ;THEY WERE
0110 D626               SUI     26H     ;WAS IT &?
0112 CA9601             JZ      TYPE    ;YES,SO DISPLAY NEXT LINE AND
                                        ;RETURN FROM TYPE TO TLOOP
0115 3C                 INR     A       ;WAS IT % (25H)?
0116 CA2B01             JZ      PAD     ;YES,GO PAD AND RETURN TO TLOOP
                                        ;FROM THE PAD ROUTINE
0119 3C                 INR     A       ;WAS IT $ (24H)?
011A CA5A01             JZ      EXEC0   ;YES,EXECUTE THE USER PROGRAM
011D 3C                 INR     A       ;WAS IT # (23H)?
011E CA0001             JZ      START   ;YES,BACK TO START-STACK WILL BE
                                        ;CLEANED UP AUTOMATICALLY BY THE
                                        ;FIRST TWO INSTRUCTIONS!

                ;IF IT WAS NONE OF THESE IT MUST HAVE BEEN A CHARACTER
                ;TO PUT INTO THE USER PROGRAM.BEFORE WE CAN DO THIS WE
                ;MUST RESTORE THE CHARACTER THE WAY IT WAS WHEN WE FIRST
                ;GOT IT FROM THE ROUTINE CI

0121 C623               ADI     23H     ;RESTORE IT AS IT WAS
0123 77                 MOV     M,A     ;PUT IT INTO USER PROGRAM
0124 23                 INX     H       ;AND POINT TO NEXT SEQUENTIAL
                                        ;LOCATION IN USER PROGRAM AREA

                ;IF THAT WAS A CARRAGE RETURN (CR) WE JUST PUT AWAY WE
                ; NOW SEND A LINE FEED (LF) TO KEEP THE TERMINAL HAPPY
0125 FE0D               CPI     0DH     ;WAS IT CR?
0127 CC5301             CZ      PLF     ;IF SO TYPE A LF
                                        ;NOTE-THIS MUST BE A CALL
012A C9                 RET             ;IN EITHER CASE BACK TO TLOOP
                                        ;FOR THE NEXT CHARACTER

                ;THAT IS THE END OF THE MASTER TEXT EDIT ROUTINE.

                ;NOW FOR THE MAIN SUBROUTINES THAT THE MASTER
                ;ROUTINE CALLS. IN CASE YOU WISH TO MODIFY THE
                ;ABOVE NOTE THAT THE INTERPRETER (EXEC) ALSO
                ;USES THE COM ROUTINE.THE TYPE SUBROUTINE WILL
                ;BE FOUND AS PART OF THE INTERPRETER


                ;THE PAD ROUTINE-IT MUST PRECEED THE COM ROUTINE
                ;AS IT 'FALLS THROUGH' TO IT!

012B 1680       PAD:    MVI     D,80H   ;SET COMMON ROUTINE FLAG = 'PAD'

                ;THE COMMON (COM) ROUTINE. IF ENTERED WITH D =OR> 80H IT
                ;PADS THE USER PROGRAM AREA WITH NULLS. IT STARTS AT THE
                ;LOCATION POINTED TO BY H&L AND PADS UP TO BUT NOT
                ;INCLUDING THE FIRST LOCATION IN WHICH IT FINDS A CR. A
                ;CR AND LF ARE THEN SENT TO THE TERMINAL. IF ON ENTRY D
                ;< 80H IT TYPES THE CONTENTS OF THE USER PROGRAM AREA
                ;FROM THE LOCATION POINTED TO BY H&L. IT TYPES THE TEXT
                ;UNTIL A CR IS FOUND.THIS CR IS TYPED AND THEN A LF IS
                ;ALSO TYPED. IN ALL CASES THERE IS A SAFTY COUNT OF 64
                ;IN FORCE-IF 64 CHARACTERS HAVE BEEN TYPED OR PADDED
                ;WITHOUT A CR BEING FOUND A '?' IS TYPED AND THE
                ;ROUTINE ABORTS TO START

012D C5         COM:    PUSH    B       ;SAVE B&C ON THE STACK
                                        ;TO MAKE SOME ROOM
012E 0640               MVI     B,40H   ;SET A SAFETY COUNT OF 64
0130 7E         COM1:   MOV     A,M     ;GET A CHARACTER TO A
0131 FE0D               CPI     0DH     ;IS IT CR?
0133 CA4E01             JZ      COM2    ;JOB FINISHED WHEN WE FIND A CR
                                        ;SO CLEAN UP BY TYPING BOTH A
                                        ;CR AND A LF
0136 4F                 MOV     C,A     ;COPY A INTO C IN CASE WE ARE
                                        ;TO TYPE
0137 7A                 MOV     A,D     ;GET THE FLAG
0138 07                 RLC             ;MOVE BIT 7 INTO THE CARRY FLAG
0139 D23E01             JNC     NOPAD   ;NO CARRY MEANS DON'T PAD
013C 3600               MVI     M,0     ;IF THERE IS A CARRY WE MUST
                                        ;PAD SO PUT IN ONE NULL

                ;NOTE- THE LAST INSTRUCTION DOES NOT AFFECT THE STATE
                ;OF THE CARRY FLAG SO IF THE LAST INSTRUCTION
                ;WAS DONE THE NEXT ONE WON'T BE.

013E D4F901     NOPAD:  CNC     CO      ;NO CARRY,SO TYPE CHARACTER
0141 23                 INX     H       ;POINT TO NEXT LOCATION TO TREAT
0142 05                 DCR     B       ;64 CHARACTERS YET?
0143 C23001             JNZ     COM1    ;NO,GO AND DO MORE
0146 0E3F               MVI     C,'?'   ;YES,WE HAVE A PROBLEM
0148 CDF901             CALL    CO      ;PRINT A '?'
014B C30001             JMP     START   ;AND ABORT TO START

014E 4E         COM2:   MOV     C,M     ;PUT THE CR IN C
014F 23                 INX     H       ;MOVE OVER THE CR
0150 CDF901             CALL    CO      ;PRINT IT
0153 0E0A       PLF:    MVI     C,0AH   ;LOAD C WITH A LF
0155 CDF901             CALL    CO      ;PRINT IT
0158 C1                 POP     B       ;RESTORE B&C
0159 C9                 RET             ;AND BACK TO WHOEVER CALLED US

                ;END OF TEXT EDITOR AND IT'S MAIN SUPPORT ROUTINES


                ;THIS IS THE START OF THE INTERPRETER MAIN ROUTINE

                ;WHEN WE ARRIVE AT EXEC WE HAVE THE ADDRESS OF TLOOP
                ;ON THE TOP OF THE STACK AND THE ADDRESS AT WHICH WE
                ;WISH TO START EXECUTION OF THE USER PROGRAM IN H&L
                ;ALL THE OTHER REGISTERS ARE AS YET UNDEFINED,BUT
                ;WILL HAVE THE FOLLOWING USES:-
                ;D = MARKER COUNTER OR PRINT/PAD FLAG
                ;E = USER COUNTER
                ;B = RESULT OF LAST MATCH (AF=YES,0=NO)
                ;C = LAST CHARACTER THAT WAS INPUT IN RESPONSE TO AN
                ;    ACCEPT COMMAND

                ;FIRST WE LOSE THE ADDRESS OF TLOOP FROM THE TOP
                ;OF THE STACK AS WE NO LONGER NEED IT.

015A D1         EXEC0:  POP     D       ;THERE,LOST IT

                ;NOW WE PUT THE ADDRESS OF EXEC ONTO THE TOP OF THE
                ;STACK WITHOUT ALTERING ANY REGISTERS. THIS LETS US
                ;RETURN TO EXEC BY A SIMPLE RETURN INSTRUCTION.
                ;OF COURSE WE MUST DO THIS AGAIN EACH TIME WE RETURN
                ;TO EXEC AS WE 'USE UP' THE ADDRESS GETTING THERE.

015B E5         EXEC:   PUSH    H       ;PUT H&L ONTO STACK
015C 215B01             LXI     H,EXEC  ;PUT ADDRESS OF EXEC IN H&L
015F E3                 XTHL            ;SWAP H&L WITH TOP OF STACK
0160 7E                 MOV     A,M     ;GET A CHARACTER FROM THE USER
                                        ;PROGRAM AREA
0161 23                 INX     H       ;POINT TO NEXT CHARACTER FOR
                                        ;NEXT TIME ROUND
0162 FE5A               CPI     'Z'     ;NO STATEMENT STARTS WITH
                                        ;Z OR ANYTHING BEYOND THAT
0164 D28C01             JNC     ERROR   ;SO IF IT'S > OR = 'Z',IT'S AN
                                        ;ERROR!
0167 D641               SUI     'A'     ;SUBTRACT ASCII A
0169 D8                 RC              ;IGNORE IT IF IT WAS < 'A'

                ;NOTE,IT COULD HAVE BEEN A CR,A MARKER (*),
                ;AN OPTIONAL STATEMENT DELINEATOR OR EVEN
                ;A NUMBER PUT IN BY SOME OTHER LINE ORIENTED TEXT
                ;EDITOR.IN ANY CASE WE DON'T WANT TO KNOW ABOUT IT
                ;AT THE MOMENT SO WE JUST JUMP OVER IT.

                ;WE NOW HAVE THE INDENTIFYING KEY LETTER FROM WHICH
                ;ASCII 'A' HAS BEEN SUBTRACTED IN REGISTER A. WE LOOK
                ;UP IN A TABLE STARTING AT TBASE TO FIND THE LEAST
                ;SIGNIFICANT BYTE OF THE ADDRESS OF THE SUBROUTINE
                ;WHICH PERFORMS THE ACTUAL STATEMENT. AS THE WHOLE
                ;INTERPRETER FITS IN 256 BYTES WE ALREADY KNOW THE
                ;MOST SIGNIFICANT BYTE

016A E5                 PUSH    H       ;WE NEED A LITTLE ROOM
016B 217301             LXI     H,TBASE ;ADDRESS OF FIRST ENTRY
                                        ;IN TABLE TO H&L
016E 85                 ADD     L       ;ADD L TO THE KEY LETTER IN A
                                        ;WHICH IS IN THE RANGE FROM
                                        ;A=0 TO Y=25
016F 6F                 MOV     L,A     ;NOW H&L HAS THE ADDRESS
                                        ;OF THE ENTRY WE WANT
0170 6E                 MOV     L,M     ;NOW H&L HAS THE ADDRESS OF
                                        ;THE SUBROUTINE WE WANT
0171 E3                 XTHL            ;PUT THIS ON TO THE TOP OF
                                        ;THE STACK AND RESTORE THE
                                        ;ORIGIONAL H&L ALL AT ONCE
0172 C9                 RET             ;AND OFF TO THE ADDRESS WHICH
                                        ;WE JUST PUT ON THE STACK

                ;THE NEXT 25 BYTES CONTAIN THE LEAST SIGNIFICANT
                ;BYTE OF THE SUBROUTINES THAT ACTUALLY PERFORM
                ;THE ACTION REQUIRED TO DO THE STATEMENT. THEY ARE
                ;IN ORDER,A (ACCEPT) FIRST TO Y (YES) LAST
                ;THE FORM MOD 256 WHICH APPEARS BELOW IS A WAY
                ;OF TELLING MY ASSEMBLER TO ONLY USE THE LEAST
                ;SIGIFICANT 8 BITS OF THE ADDRESS

0173 F3         TBASE:  DB      CI      MOD 256 ;A ENTRY
0174 8C                 DB      ERROR   MOD 256 ;B ENTRY
0175 C1                 DB      SKIP    MOD 256 ;C ENTRY
0176 CF                 DB      DEC     MOD 256 ;D ENTRY
0177 91                 DB      END1    MOD 256 ;E ENTRY
0178 8C                 DB      ERROR   MOD 256 ;F ENTRY
0179 DA                 DB      GET     MOD 256 ;G ENTRY
017A 8C                 DB      ERROR   MOD 256 ;H ENTRY
017B D1                 DB      INC     MOD 256 ;I ENTRY
017C 9E                 DB      JUMP    MOD 256 ;J ENTRY
017D 03                 DB      KEEP    MOD 256 ;K ENTRY
017E CC                 DB      LCNTR   MOD 256 ;L ENTRY
017F B4                 DB      MATCH   MOD 256 ;M ENTRY
0180 BE                 DB      TESTN   MOD 256 ;N ENTRY
0181 8C                 DB      ERROR   MOD 256 ;O ENTRY
0182 F9                 DB      CO      MOD 256 ;P ENTRY
0183 8C                 DB      ERROR   MOD 256 ;Q ENTRY
0184 B0                 DB      RETN    MOD 256 ;R ENTRY
0185 98                 DB      SUBR    MOD 256 ;S ENTRY
0186 96                 DB      TYPE    MOD 256 ;T ENTRY
0187 F0                 DB      USER    MOD 256 ;U ENTRY
0188 8C                 DB      ERROR   MOD 256 ;V ENTRY
0189 8C                 DB      ERROR   MOD 256 ;W ENTRY
```

```
018A C8              DB      EXCH     MOD 256 ;X ENTRY
018B 80              DB      TESTY    MOD 256 ;Y ENTRY

                ;END OF LOOK UP TABLE

                ;NOW FOR THE SUBROUTINES CALLED BY EXEC. THESE ARE -
                ;END1,ERROR,EXCH,GET,INC,JUMP,KEEP,LCNTR,MATCH,
                ;RETN,SUBR,TESTN,TESTY AND TYPE.

                ;NOTE THE ORDER OF THESE NEXT THREE SUBROUTINES AS
                ;THEY 'FALL THROUGH' FROM ONE TO ANOTHER.

                ;IF WE ARRIVE AT EITHER ERROR OR END1 WE HAVE FINISHED
                ;EXECUTING THE USER PROGRAM AND MUST GO BACK TO THE
                ;START. IN THIS CASE THE CONTENTS OF D&E ARE NO LONGER
                ;OF ANY IMPORTANCE

018C 0E3F       ERROR   MVI     C,'?'    ;LOAD A QUESTION MARK
018E CDF901             CALL    CO       ;PRINT IT AS WE HAVE AN ERROR
0191 2B         END1    DCX     H        ;BACK UP TO SHOW THE CHARACTER
                                         ;WHICH HAS CAUSED THE TROUBLE
0192 110001             LXI     D,START  ;GET READY TO GO BACK TO START
0195 D5                 PUSH    D        ;START ADDRESS TO TOP OF STACK
0196 1600       TYPE    MVI     D,0      ;SET COM FLAG TO TYPE
0198 C32D01             JMP     COM      ;GO AND PRINT LINE,THE RETURN
                                         ;AT THE END OF COM SENDS US
                                         ;BACK TO EXEC UNLESS WE CAME
                                         ;HERE BY WAY OF ERROR OR END1
                                         ;WHEN IT SENDS US TO START

019B 220002     SUBR    SHLD    RADR     ;SAVE ADDRESS OF THIS MARKER
                                         ;FOR USE LATER AS THE RETURN
                                         ;ADDRESS.AS IT IS A NUMBER
                                         ;EXEC WILL SKIP OVER IT FOR
                                         ;US WHEN WE COME BACK

                ;ONCE THE RETURN ADDRESS HAS BEEN SAVED,THE REST OF THE
                ;SUBROUTINE OPERATION IS IDENTICAL TO A JUMP.
                ;SO "SUBR" FALLS THROUGH TO "JUMP" AT THIS POINT

                ;FIRST WE USE VCOM TO GET THE MARKER NUMBER TO A & TO
                ;CHANGE IT FROM AN ASCII NUMBER TO BINARY AND THEN
                ;TO SUBTRACT ONE -----r THE RESULT FROM ALL THIS
                ;IS THE MARKER COUNTER WHICH VCOM PUTS IN D FOR US
                ;VCOM CHECKS TO SEE THAT THE MARKER NUMBER IS NOT
                ;<1 OR >9 AND GOES TO ERROR IF IT IS.
                ;VCOM DOES MORE THAN THIS AS IT IS ALSO USED BY THE
                ;GET AND KEEP ROUTINES,BUT THE FACT THAT IT CORRUPTS
                ;H&L IS OF NO CONCERN TO US AS WE ARE ABOUT TO RELOAD
                ;THEM ANYWAY

019E CDE101     JUMP:   CALL    VCOM     ;GET THE MARKER COUNTER TO D
01A1 211502             LXI     H,UPROG  ;START LOOKING FOR MARKERS FROM
                                         ;BEGINNING OF THE USER PROGRAM
01A4 3E2A               MVI     A,'*'    ;THIS IS WHAT WE ARE LOOKING FOR
01A6 BE         JLOOP:  CMP     M        ;FOUND ONE AT THIS USER ADDRESS?
01A7 23                 INX     H        ;POINT TO NEXT USER ADDRESS
01A8 C2A601             JNZ     JLOOP    ;NO WE HAVEN'T,KEEP LOOKING

                ;AH,WE HAVE FOUND ONE - BUT IS IT THE CORRECT ONE?

01AB 15                 DCR     D        ;IF IT IS,D WILL NOW BE NEGATIVE
01AC F2A601             JP      JLOOP    ;BOTHER,WRONG ONE - KEEP LOOKING
01AF C9                 RET              ;GOT IT,NOW CARRY ON EXECUTION
                                         ;FROM WHERE WE FOUND IT

01B0 2A0002     RETN:   LHLD    RADR     ;WHERE WERE WE IN THE USER'S
                                         ;PROGRAM WHEN WE CAME TO DO THIS
                                         ;SUBROUTINE?
01B3 C9                 RET              ;CARRY ON EXECUTION FROM THERE

01B4 7E         MATCH:  MOV     A,M      ;GET CORRECT ANSWER TO A
01B5 23                 INX     H        ;POINT TO NEXT USER COMMAND
01B6 91                 SUB     C        ;ZERO RESULT IF THEY MATCHED
01B7 06AF               MVI     B,0AFH   ;SET "YES" FLAG IN CASE THEY DID
01B9 C8                 RZ               ;THEY DID - CARRY ON
01BA 0600               MVI     B,0      ;THEY DID NOT - SET "NO" FLAG
01BC C9                 RET              ;AND NOW CARRY ON

01BD 3E         TESTY   DB      3EH      ;IF WE COME IN HERE THIS PLUS
                                         ;THE NEXT BYTE WILL BE READ BY
                                         ;THE PROCESSOR AS A SINGLE
                                         ;INSTRUCTION TO LOAD AF (HEX) TO
                                         ;REGISTER A,I.E. AS MVI A,0AFH
01BE AF         TESTN:  XRA     A        ;IF WE COME IN HERE WE JUST
                                         ;CLEAR A TO ZERO
01BF B8                 CMP     B        ;TEST RESULT OF LAST MATCH (IN
                                         ;B) WITH A.IF WE ARE LOOKING FOR
                                         ;"YES" A WILL BE AF IF WE ARE
                                         ;LOOKING FOR "NO" A WILL BE 0
01C0 C8                 RZ               ;0 MEANS TEST PASSED SO CARRY ON
                                         ;WITH THE REST OF THIS USER LINE

                ;IF THE TEST WAS FAILED WE MUST SKIP TO THE END OF THIS
                ;USER LINE AND CARRY ON EXECUTION FROM THERE. WE USE
                ;THE JLOOP ROUTINE TO SEARCH FOR THE CR AT THE END OF
                ;THIS USER LINE AFTER FIRST SETTING UP TO FIND ONLY
                ;ONE CR.

01C1 3E0D       SKIP:   MVI     A,0DH    ;0D IS ASCII FOR CR-WHAT WE WANT
01C3 1600               MVI     D,0      ;SO WE STOP AT THE FIRST ONE
01C5 C3A601             JMP     JLOOP    ;JLOOP DOES THE REST AND WE
                                         ;RETURN FROM THE END OF IT TO
                                         ;EXEC TO CARRY ON

01C8 7B         EXCH:   MOV     A,E      ;PUT E INTO A
01C9 59                 MOV     E,C      ;PUT C INTO E
01CA 4F                 MOV     C,A      ;AND A INTO C
01CB C9                 RET              ;AND CARRY ON

01CC 5E         LCNTR   MOV     E,M      ;LOAD THE COUNTER
01CD 23                 INX     H        ;MOVE OVER THAT CHARACTER
01CE C9                 RET              ;BACK TO EXEC FOR MORE

01CF 1D         DEC:    DCR     E        ;DECREMENT COUNTER
01D0 C9                 RET              ;BACK TO EXEC FOR MORE

01D1 1C         INC:    INR     E        ;INCREMENT COUNTER
01D2 C9                 RET              ;AND BACK TO EXEC FOR MORE

01D3 E5         KEEP:   PUSH    H        ;WE NEED SOME ROOM
01D4 CDE101             CALL    VCOM     ;FIND ADDRESS WHERE TO
01D7 71                 MOV     M,C      ;  SAVE C
01D8 E1                 POP     H        ;RESTORE H&L
01D9 C9                 RET              ;AND BACK TO EXEC FOR MORE

01DA E5         GET:    PUSH    H        ;WE NEED SOME ROOM
01DB CDE101             CALL    VCOM     ;FIND ADDRESS WHERE TO
01DE 4E                 MOV     C,M      ;  LOAD C FROM
01DF E1                 POP     H        ;RESTORE H&L
01E0 C9                 RET              ;AND BACK TO EXEC FOR MORE

01E1 7E         VCOM:   MOV     A,M      ;WHICH VARIABLE OR MARKER?

                ;NOTE WE DONT SKIP OVER VARIABLE OR MARKER # AS
                ;EXEC WILL DO IT FOR US. IF WE DID,THE ERROR ROUTINE
                ;WOULD NOT SHOW US THE WHOLE OF THE OFFENDING
                ;STATEMENT!

01E2 D631               SUI     '1'      ;CONVERT TO BINARY-1

                ;IF VARIABLE OR MARKER # WAS <'1' IT WILL NOW BE 256-#
                ;AND THE CHECK IN THE NEXT LINE WILL PICK IT UP TOO

01E4 FE09               CPI     9        ;>9 OR <1?IMPOSSIBLE!
01E6 D28C01             JNC     ERROR    ;SO GO AND COMPLAIN

                ;THE NEXT INSTRUCTION IS ONLY OF USE IF WE ARE
                ;WORKING OUT THE MARKER COUNTER - BUT IT DOES NOT
                ;DO ANY HARM IF WE ARE WORKING OUT A VARIABLE
                ;ADDRESS

01E9 57                 MOV     D,A      ;PUT IT IN D IN CASE

                ;NOW TO WORK OUT THE ADDRESS OF THE PARTICULAR
                ;STORAGE LOCATION IDENTIFIED BY THE BINARY NUMBER IN A
                ;THIS IS OF NO USE (OR HARM) IF WE ARE WORKING OUT
                ;THE MARKER COUNTER!

01EA 210202             LXI     H,VBASE  ;POINT TO STORAGE LOCATION 1
01ED 85                 ADD     L        ;ADD THE INDEX FROM A TO THE
                                         ;LEAST SIGNIFICANT BYTE OF THE
                                         ;BASE ADDRESS WHICH IS IN L
01EE 6F                 MOV     L,A      ;AND PUT THE ANSWER INTO L SO
                                         ;H&L NOW POINT TO THE WANTED
                                         ;STORAGE LOCATION
01EF C9                 RET              ;RETURN TO WHOEVER CALLED US

                ;THE NEXT LINE PROVIDES A LINKAGE OUT OF THIS
                ;PAGE FOR THE USER STATEMENT.AT PRESENT IT IS
                ;FLAGGED AS AN ERROR. WHEN YOU USE IT CHANGE THE VALUE
                ;OF THE ADDRESS TO JUMP TO.

01F0 C38C01     USER:   JMP     ERROR    ;'U' IS NOT IMPLIMENTED YET

                ;THE NEXT FEW LINES HANDLE THE SINGLE CHARACTER
                ;INPUT AND OUTPUT. AS WRITTEN HERE THEY USE TWO OF
                ;MY MONITOR'S SUBROUTINES AND SO WILL NEED ALTERING
                ;TO SUIT YOUR SYSTEM. MI GETS A SINGLE CHARACTER
                ;FROM THE TERMINAL TO A,DOES NOT ECHO IT AND
                ;DOES NOT STRIP OFF THE PARITY BIT (BIT 7). MO
                ;PRINTS THE SINGLE CHARACTER IN C ON THE TERMINAL.
                ;YOU WILL PROBABLY HAVE TO REWRITE ALL THE CODE
                ;BETWEEN THE LINES OF DASHES
```

(Refer to page 61 for modified I/O routines).

```
                ;-----------------------------------------------------
3803            MI      EQU     3803H    ;ADDRESS OF MY MONITOR INPUT
                                         ;ROUTINE
3809            MO      EQU     3809H    ;ADDRESS OF MY MONITOR OUTPUT
                                         ;ROUTINE

01F3 CD0338     CI      CALL    MI       ;GET A CHARACTER
01F6 E67F               ANI     7FH      ;STRIP OFF PARITY BIT
01F8 4F                 MOV     C,A      ;PUT IN C READY TO ECHO
01F9 CD0938     CO      CALL    MO       ;PRINT CHARACTER IN C
01FC 79                 MOV     A,C      ;COPY IT BACK TO A
01FD C9                 RET              ;AND RETURN TO WHOEVER
                                         ;CALLED US
                ;-----------------------------------------------------

                ;END OF THE PROGRAM. EVERYTHING UP TO HERE COULD BE
                ;KEPT IN PROM IF YOU WISH. THERE ARE STILL TWO SPARE
                ;BYTES IN THE 256 BYTE PROGRAM AREA IN CASE YOUR CI
                ;AND CO ROUTINES ARE LONGER THAN NINE

                ;EVERYTHING FROM NOW ON MUST BE IN RAM. RESERVE
                ;SOME SPACE FOR THE USER SUBROUTINE RETURN ADDRESS,
                ;THE NINE SINGLE CHARACTER STORAGE LOCATIONS AND
                ;THE STACK.
                ;SET A NEW ORIGIN SO THAT THESE STORAGE LOCATIONS
                ;AND THE USER PROGRAM SPACE START ON A NEW PAGE

01FE                    ORG     START+100H

0200            RADR    DS      2        ;TWO BYTES FOR USER SUBROUTINE
                                         ;RETURN ADDRESS
0202            VBASE   DS      9        ;NINE BYTES FOR SINGLE
                                         ;CHARACTER STORAGE LOCATIONS
020B                    DS      10       ;TEN BYTES FOR THE STACK
0215            UPROG   EQU     $        ;USER PROGRAM AREA STARTS HERE
0000                    END
SYMBOL TABLE:

CI      01F3     CO      01F9     COM     012D     COM1    0130
COM2    014E     DEC     01CF     END1    0191     ERROR   018C
EXCH    01C8     EXEC    0158     EXEC0   015A     GET     01DA
INC     01D1     JLOOP   01A6     JUMP    019E     KEEP    01D3
LCNTR   01CC     MATCH   01B4     MI      3803     MO      3809
NOPAD   013E     PAD     0128     PLF     0153     RADR    0200
RETN    01B0     SKIP    01C1     START   0100     SUBR    019B
TBASE   0173     TESTN   01BE     TESTY   01BD     TLOOP   0104
TYPE    0196     UPROG   0215     USER    01F0     VBASE   0202
VCOM    01E1
```

Write your modified I/O routines here.

to build a random number generator into Simple, but the form used in this program makes it hard to predict ahead from turn to turn.

The sixth and last example is really a pair of programs, an enciphering program and a deciphering program. For a moment diverting slightly into ciphers, the simplest cipher is one in which every letter in the plain text message is swapped for another. A letter (say e) is always replaced by the same alternative letter (say k). In other words, we always use the same enciphering alphabet. The characteristic features of the language are not altered — for example, in English, 'e' is the most common letter, with a, o, i, t, n, r, s and h also occurring frequently. On the other hand, j, h, q, x and z occur rarely. If a letter follows a vowel four-fifths of the time and only appears before it one-fifth, the letter is probably n. The combination th and he are common, but ht and eh are rare. From these and other characteristic relationships the cipher can quite readily be broken.

In our cipher *we change the enciphering alphabet after every character* — the previous cipher letter guiding us to which of the many possible enciphering alphabets to use next. Our program is an electronic version of a combination of the twin rotating discs described by Leon Battista Alberti in 1466

```
&    T WHAT IS THE KEY?
&    A,X, C ORIGIONAL KEY TO COUNTER
&    T
&    T START TEXT,TO GET A NEW LINE TYPE A +,
&    T AT END OF TEXT TYPE A FULL STOP.
&    T AFTER EACH CHARACTER YOU ENTER I WILL
&    T GIVE YOU THE ENCIPHERED CHARACTER.
&    T USE AN @ SYMBOL TO REPRESENT THE
&    T SPACE BETWEEN WORDS.
&    C START OF OUTER LOOP
&1*  A, C CHECK FOR A NEW LINE
&    M+,YT
&    YJ1
&    C CHECK FOR END OF TEXT
&    M.,YT
&    YE
&    C CYCLE THE COUNTER
&2*  X,M0,X,D,YLZ
&    C REDUCE THE INPUT CHARACTER TO 0
&    M0,X,D,X,NJ2
&    C PRINT OUTPUT CHARACTER
&    X,P
&    C PRINT A SPACE
&    L ,X,P
&    J1, C GO AND ENCIPHER MORE

&    T WHAT IS THE KEY?
&    A,X, C ORIGIONAL KEY TO COUNTER
&    T
&    T START TEXT,TO GET A NEW LINE TYPE A +
&    T AT END OF TEXT TYPE A FULL STOP.
&    T AFTER EACH CHARACTER YOU ENTER I WILL
&    T GIVE YOU THE DECIPHERED CHARACTER.
&    T TO MARK THE SPACE BETWEEN WORDS USE
&    T AN @ SYMBOL.
&    C START OF OUTER LOOP
&1*  A, C CHECK FOR A NEW LINE
&    M+,YT
&    YJ1
&    C CHECK FOR END OF TEXT
&    M.,YT
&    YE
&    C SAVE INPUT CHARACTER FOR NEXT TIME
&    K1
&    C CYCLE THE COUNTER
&2*  X,M0,X,D,YLZ
&    C REDUCE INPUT CHARACTER TO 0
&    M0,X,D,X,NJ2
&    C PRINT OUTPUT CHARACTER
&    X,P,X
&    C PRINT A BLANK
&    L ,X,P
&    C RESTORE LAST CHARACTER INPUT TO COUNTER
&    C AND THEN GO AND DECIPHER MORE
&    G1,X,J1
```

```
11 WHAT IS THE KEY?
A
 START TEXT,TO GET A NEW LINE TYPE A +,
 AT END OF TEXT TYPE A FULL STOP.
 AFTER EACH CHARACTER YOU ENTER I WILL
 GIVE YOU THE ENCIPHERED CHARACTER.
 USE AN @ SYMBOL TO REPRESENT THE
 SPACE BETWEEN WORDS
EV L9 EN C: I@ RH DS N4 IE CI S8 @P TX D8 DN A< Y= @W +
I= NI IO E9 RA WM B, IA IR D2 N> AW L: .
E
```

Program 6. Enciphering and deciphering programs.

Hex Value	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35
ASCII Character		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	3	4	5
Counter Value	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5
Simple Usage	✓	✓	✓	T	T	T	T	✓	✓	✓	M	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Hex Value	36	37	38	39	3A	3B	3C	3D	3E	3F	40	41	42	43	44	45	46	47	48	49	4A	4B
ASCII Character	6	7	8	9	:	;	<	=	>	?	@	A	B	C	D	E	F	G	H	I	J	K
Counter Value	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Simple Usage	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	K	I	K	K	K	I	K	I	K	K	K

Hex Value	4C	4D	4E	4F	50	51	52	53	54	55	56	57	58	59	5A
ASCII Character	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Counter Value	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Simple Usage	K	K	K	I	K	I	K	K	K	K	I	I	K	K	I

Table 1. The ASCII character set between 20 (Hex) and 5A (Hex), equivalent numeric value in the counter and usage in Simple. (✓= may be used as a seperator between statements. T = text editor command. K = used key letter. M = marker. I = illegal key letter at present.)

and the auto key proposed by Blaise de Vigenere in 1586. A letter may come out as anything — even itself, but rarely comes out the same way twice. The possible characters (both input and output) are the characters in the ASCII character set, between ϕ and Z inclusive (see table 1). As this does not include a space we use an @ for a space. In order to decipher a message, the initial key (which can be any character from this range) must be known — so it is added as the first character of the enciphered text. As you see the phrase Electronics Today International comes out as the unpronounceable

'AV9N:@HS4E18RX8N<=W=109AM;AR2>W:'.

By using the various Simple statements you can build up bigger and more powerful programs than those shown here. For example, the power that comes from being able to jump (or subroutine) into and out of multiple conditional match statements on a single line has not been explored at all. Try writing a few programs that use a number of logic decisions; you will be surprised how much Simple can do — and remember you can add on extra statements you might want that I have missed out. In the meantime for those of you who enjoy small puzzles you may care to decipher the following message TWA02L7BN8RS38RYCXGX3M3:TϕANX;P.

The answer to this will be given at the end of the article.

Simple

SIMPLE — a small interpreter for microprocessors. Part 2.

Dr. Tim Hendtlass, Applied Physics Department, R.M.I.T.

HAVING DEALT WITH the simple language we now turn to the 8080 machine language program given in listing one which interprets and executes any program given to it in Simple language. No doubt your first reaction is surprise that a source listing which produces a mere 254 bytes of object (machine) code should be so long. The main reason for this is that the listing is very heavily commented. Only those lines which have some numbers at the very left-hand side of the line contain anything other than a comment. A comment is anything on a line to the right of a semi-colon (;), the assembler pays no attention to comments other than to cause them to be faithfully printed out on the final output. You and I are the sole reasons for these comments, they help us find our way through the program and understand how it works. When I first started writing programs I was lazy and did not comment them, or at least not very much. Returning to a Fast Fourier Transform program two years later to adapt it, I discovered a bitter truth — I just could not understand it. An uncommented source listing is as useful as a complete circuit diagram without any component values, wave-forms or labels of any kind.

So, since the design goal for Simple was to produce an interpreter that others could use, understand and especially modify, the source listing is very heavily commented. The listing has one complete instruction per line in general and each line is commented. In addition each block of instructions is also commented. The other items in a line of output from the assembler can be identified by reference to the line almost at the beginning of the listing which has 0100 at the far left-hand side. All lines which contain more than just comments have first of all a four hex digit address printed. This is the address of the first byte of the machine code for the instruction given further across the line. Next to this address are two, four or six hex characters which made up the one, two or three bytes of machine code for the instruction. After them comes an optional entry, a label (in this case START) — this is to allow symbolic references to addresses whose actual value is not known until the assembly is done. A label, if there is one, must end with a colon (:). Next comes the actual instruction, first the key part and then the optional parameters (if any). Finally comes the comment described earlier. With the Simple language description, many readers with a reasonable knowledge of 8080 mnemonics will now be able to work through the source listing with only minor difficulty at one or two places. However, to help those not so familiar with machine code program listings, let us now take a guided tour of the Simple interpreter.

A guided tour of the Simple interpreter

All the addresses below refer to the source listing given in this article which has been assembled to start at location 1ØØ Hex. All addresses and the machine code are given in hex. The Simple interpreter can be separated into two parts, the small text editor used to enter a Simple language program into memory and the actual interpreter which executes a user program once it has been entered.

The text editor

The text editor occupies from 1ØØ to 159 hex inclusive and is complete in itself except for the single character input and output routines CI and CO. The H and L registers are used together as a pointer to some place in the user program storage area. Unless the character entered is one of the five special characters (←, &, %, $, or #), it is put into memory at the current position pointed to by H and L. It replaces what was there and H and L are then incremented to point to the next sequential location in the user program area.

When we first come into Simple no register can be assumed to have any particular value and so first of all (1ØØ, to 1Ø3) H and L and the stack pointer are both loaded with the address of the start of the user programme area. On the 8080 the stack pointer is decremented before an item is placed on the stack so that the stack starts at Ø215 and works down in memory whereas the user program area starts at Ø215 and works up. Note that in 8080 machine code and low byte of an address is sorted before the high byte.

After each character has been processed (except $) we will want to come back to TLOOP for the next character. If we place the address of TLOOP onto the stack just as a subroutine call would have done, we can return to TLOOP by an 8080 RETURN instruction just as if we were returning from a subroutine. This saves valuable space compared to using a JUMP instruction and is the reason for the instructions from 1Ø4 to 1Ø7.

The rest of the text editor is straight forward except for the COM routine and the instruction at 127. These will be covered shortly. Remember though that the character in routine (CI) must also echo the character, I have done this by letting CI 'fall through' to the character out (CO) routine (see 1F3 to 1FD). Note this and the other requirements that CI and CO must meet (given in the listing just after 1FØ) as you will also certainly have to re-write the CI and CO code to suit your system. There are still 2 spare bytes in this page in case your routines are longer than mine.

The Interpreter

The interpreter's main routine starts at EXEC at 15BH and causes the next statement (i.e. the one pointed to by the H and L registers) to be processed. After any statement is processed we must come back to EXEC for the next one, unless we just processed an END statement or found an error. In these latter cases we go back to START and into the test editor again. We use the same technique we used in the text editor to get us easily back to TLOOP in the interpreter, only this time, of course, we put the address of EXEC onto the top of the stack as this is where we want to return to. When we put the EXEC address onto the stack we must not alter any of the 8080's registers, as in the interpreter, unlike the text editor, all of them are used. This takes three instructions and five bytes (15BH to 15FH) compared to the two instructions and four bytes used to get the TLOOP address on the stack.

We then get the key letter of the statement from memory and look to see what kind of statement we have to process, checking to see that it is indeed a legal letter. If the 'letter' comes before 'A' in the ASCII alphabet it may not be an error as it could be a carriage return, marker (*) or a number, all of which are legal, but none of which we want to know about at present. So we skip over it and get the next letter. The fact that EXEC ignores all characters before ASCII 'A' is important and we make considerable use of this later.

Having got an apparently legal key letter (at 16AH) we look up in a table starting at 173H to find the address of the sub-routine which processes this particular kind of statement. Since the whole of SIMPLE fits into one 256 byte page we only have to look up the least significant byte of the address in the table as the most significant byte is the same throughout. The rest of the machine code consists of the subroutines to process the different statements, all of which occur after the address at which the table is stored — a feature that you may find useful if you decide to expand SIMPLE (see later). Some apparently legal key letters are not actually used, the address that all of these send us to is the address of the ERROR subroutine.

The Interpreter's Subroutines

The subroutines called by the interpreter are fairly straight forward except perhaps TESTY and TESTN which will be covered in a moment. Note that VCOM is used by the KEEP, GET, JUMP and SUBROUTINE routines and it does some more error checking to see that the variable or marker number is a digit between 1 and 9 and reports an error if it is not. One possible error that is not checked for is to see that, if you try to jump to the seventh marker (for example), there are seven *'s in your program. If there are less than seven the Simple interpreter will race off through memory looking for the seventh marker and then try to carry on. The result should be so odd that you will know you have made an error! The TYPE sub-routine uses the COM routine of the test editor, but otherwise the interpreter's sub-routines are complete in themselves.

Three Points Of Special Note

Simple uses subroutines that 'fall through' from one to another and an example of this can be seen in the COM routine, which starts at 12DH. Following through this sub-routine you will see that it finally ends at 159H and that the last thing it does is to print a line feed on the terminal. As we wish at one point (127H) to print a line feed, but not to do all the rest of COM, we come in at PLF (153H). In this case there is one difficulty; way up at the start of COM we push the contents of the B and C registers on to the stack and this is normally balanced by the pop just before the final return instruction. One of the subtle bits of coding in SIMPLE concerns how we cope with this POP when we come in at PLF and thus have not done the PUSH. Unelss we take some corrective action we would POP our return address to B and C and immediately get lost. This is why the instruction at 127H (when we go to PLF) *must* be a CALL not a JUMP. Using a call pushes an extra address onto the stack (the address of the RET instruction at Ø12AH) which is really unwanted information. However, this 'unwanted' information is thrown away by the POP B instruction which is itself unwanted when we come in at PLF. So the net result is that everything balances as it should.

Another special point occurs at 1BFH in the interpreter proper. At this point we want to test the state of the YES/NO flag set by the last match statement in Simple. If we are looking for a YES answer we come in at TESTY, if for a NO answer at TESTN. The actual test is carried out at 1C1H, but before this we have to preload A with the desired answer. If we come in at TESTN this is no problem, we exclusive — or the accumulator with itself and thus clear it (NO is represented by zero). However, if we come in at TESTY the processor reads 3EH which is the op-code for MVI A, . . . and so it expects the next byte to be what it is supposed to load into A. Thus now the AFH at 1CØ is taken as data and not as an instruction. The only reasonthat a YES state in the match flag is represented by the rather unusual value of AFH is the fact that Intel chose to use AFH as the machine code for XRA A. Using this way of loading the A register to 'YES' saves several bytes.

The final point of special note again concerns the COM routine. This is used both by the TYPE routine and the PAD routine. In each of these routines it is necessary to take characters one by one from the user program area until a carriage return is found. The only difference is that in one case we type the character and in the other we replace it by a null. We use the D register as a flag to tell us which routine is to be done; this is treated (137H, 138H) in such a way as to set the carry flag if we are to pad and reset it if we are to print. Use is made of the fact that not all instructions alter the state of the carry flag, in particular the 'MVI M,Ø' at 13 CH does not. The carry must have been set at 13CH (otherwise we would have taken the jump at 139H) and is still set when we arrive at 13EH from 13CH. Therefore in this case we do not make the call to print the character at 13EH. Only if the carry was not set at 139H do we make this call at 13EH. Thus we either type or pad depending on the state of the carry flag, but never both. You will see that VCOM at 1E1H is also used by two different types of routines and so has been written in such a way as to do two nearly identical tasks at once. Shared routines have been used as Simple has been written to minimise its size rather than to optimise its speed.

Modifying Simple — Without An Assembler

As stated in the introduction, Simple is designed to allow you to add on statements you would like that I have missed out. There are several ways to do this, the simplest is to use the 'U' or USER statement. To use this the two byte address at 1F1H and IF2H must be altered from the error routine address to the address of the start of your routine. One of the reasons for supplying a version of the Simple interpreter starting at 1ØØH was to leave the memory below 1ØØH as one possible memory area for you to experiment in without any risk of getting tangled up in either the stack or the user program area. As long as normal stack operations are followed in your routine (as many 'pushes' as 'pops' for example) and your routine ends with a return instruction it will correctly take you back to EXEC for the next statement. Only the A register may be altered by your routine although, of course, you may temporarily save other registers on the stack. Your routine may have put up to four extra 16-bit items on the stack at any time without exceeding the space reserved for SIMPLE's stack.

If you want to have more than one extra statement (for example a tape save and a tape load routine) there are also several ways of doing this. One way which does not require you to alter my code is to call your statements U1, U2, etc. Then your routine must check the next character after the 'U' to see which routine is needed and have a way of branching to that one. If you have access to an assembler you have more scope as it is then easy to change the origin of the program.

Modifying Simple — With An Assembler

The only reason for orgainising Simple to start on a page

Simple

boundary are for the ease of the table look up at 16A and for the convenience of anyone wanting to move the whole of SIMPLE to another page by hand as only the most significant byte of each address will need to be changed. If you have an assembler you may use the currently unused key letters (B, F, H, O, Q, V and W) by merely changing the program origin so that TBASE is at the start of a page. Then the same look up procedure will enable you to go anywhere in the page starting at TBASE. (Now you know why all sub-routines referenced from the table occur after the table!) This can be done by setting the original origin to XY00 – 73 where XY is the page you want TBASE on (e.g. to get TBASE at the start of page 2, set the origin to 18D). Now you have 73 bytes of free space on that page you can branch to – of course, you must also change the user program and storage area origin so there is no conflict. If you change the table look up procedure you can have as much space as you like, of course, but this involves modifying my code.

Modifying SIMPLE – With or Without

Naturally you may use any of the subroutines in SIMPLE as part of your new routine, whether you use an assembler or not – another reason for documenting them fairly fully in the listing. The text editor in SIMPLE is hardly the world's greatest and if you have any other text editor available you might care to use it instead – but if you decide to delete the SIMPLE text editor, remember that the interpreter proper uses the COM routine so that must stay. In defense of Simple's text editor it only accounts for 43 bytes (excluding COM). Even if yours is a line oriented text editor (i.e. requires a line number before each line) don't worry, it will work – EXEC skips over numbers at the start of a line, remember?

Simple On Other Microprocessors

If you use an 8085 or Z-80 or microprocessor Simple will run just as it does on an 8080, but if you do use a Z-80 you may use relative jumps instead of absolute (saving one byte a time) and also use the search instructions to do part of what JLOOP is doing. You also have index registers and a second register bank to use. You may even want to use the block move instruction to 'beef up' the text editor to permit inserting extra characters into an existing line.

If you use a 6800, 6502 or 2650 microprocessor you have equivalents to most of the 8080 instructions used here, but I doubt you can do any one – for – one translation. Because of the different instruction sets you may be able to do some things more simply (e.g. the table look up using index registers), but others may take longer. The 8080 instruction set is fairly register oriented, a useful feature when, as here, you can keep most of the 'running' information on the chip. A shortage of internal registers may force you to keep some of this information in extra RAM locations.

If you use a SC/MP the biggest problem you may face is the lack of an obvious external stack. P3 only lets you subroutine one deep, but if you write subroutines to handle an external stack (push, pop, call, return) you should be able to code Simple. Even though you have auto indexing going for you, I doubt you will get it into 256 bytes though.

I would be interested to hear from anyone who writes Simple for another microprocessor – it might become the subject of another article in ETI.

And Finally

Whatever you do with SIMPLE, use it, modify it or just think about it, I wish you fun – after all that is the purpose of a hobby. One last thought; if you do customise it, I suggest you add the word "Revised" to my title of 'Small Interpretive MicroProcessor Language Experiment' – then your version is SIMPLER.

Solution to the enciphering puzzle.

```
#I WHAT IS THE KEY?
I
 START TEXT,TO GET A NEW LINE TYPE A +
 AT END OF TEXT TYPE A FULL STOP
 AFTER EACH CHARACTER YOU ENTER I WILL
 GIVE YOU THE DECIPHERED CHARACTER.
 TO MARK THE SPACE BETWEEN WORDS USE
 AN @ SYMBOL
NW AE QL 2L L@ ZD BO NN 8E R@ SY 3O 8U R@ +
YS LE XE G@ XI 3T M@ 3I .S I@ 0S AI NM XP ZL FE .
E
```