A Beginner's Guide

to the

VZ200/VZ300

EDITOR ASSEMBLER

Compiled by

P.C. Schaper

CONTENTS

References Used in the Compilation of this Guide.

VZ200/300 Reference Manuals.

VZ200 Editor Assembler Instruction Manual, Dick Smith (Electronics).

VZ300 Main Unit Manual, 1983, Video Technology Ltd.

VZ300 Technical Manual, 1985, Dick Smith Management Pty Ltd.

Other References.

Borden, W. Jr., 1978, The Z80 Microcomputer Handbook, Howard W. Sams & Co. Inc., Indianapolis.

Harwood, M., (Ed.), VZ User, (A user group "magazine").

Mitchell, C., 1985, Electronics Notebook 3, Talking Electronics, Cheltenham, Vic.

Tootill, A., and Barrow, D., 1983, Z80 Machine Code for Humans, Granada, London.

Watts, L., and Wharton, M., 1983, Machine Code For Beginners, Usborne Publishing Ltd., London.

Zaks, Rodney, 1980, Programming the Z80 (second edition), Sybex Inc., USA

** Use the remaining space on this page to list any other references you find useful.

INTRODUCTION

This guide refers to the Dick Smith VZ-200 Editor Assembler V.1.2 written by Dubois and McNamara.

The instruction manual supplied with the VZ Editor Assembler, although adequate for those experienced in assembly language programming. is less than ideal for the beginner. The VZ however is a very good beginner's computer because of its low cost and the genuine computing capabilities it offers. This guide therefore is intended as an introduction to assembly language programming on the VZ200 or VZ300 computer for those who are using a VZ to learn about computers and/or computing.

This guide is based closely on the original instruction manual supplied with the VZ Editor Assembler. but the original material has been expanded and rearranged to make it more comprehensive and easier to refer to while working. Nonetheless this is still intended only as a guide to the VZ Editor Assembler rather than as a textbook on assembly language programing and those whose interest is aroused will need to refer also to one or more of the detailed references available. several of which are listed on the following page.

A more detailed section on Opcodes has been provided because of the difficulty a beginner can have in understanding the standard tables of opcodes. The Opcodes have been arranged topically as far as is possible to make it easier for a beginner to locate the Opcode that will do what he is trying to do. something which is not achieved with an alphabetical listing. Special acknowledgment is appropriate here to Collin Mitchell, whose "Machine Codes Explained" in his Electronics Notebook 3 has been invaluable in the preparation of these notes.

The emphasis throughout is intended to be on the "what" of programming rather than the "how", i.e. what to do to make something happen, rather than delving into the theory of how the computer works. Some theory is unavoidable though, and those who wish to go further will find themselves going into the nesessary theory as they progress. In the meantime this manual tries to keep the theory down to the unavoidable minimum.

The examples of programs should be of particular value to the beginner as in practice many people learn programming by starting with those written by other people rather than from a study of the computer and programming theory. The programs provided here may also be used as a foundation for a library of program 'routines' that can be incorporated into longer and more complex programs. Such a library is strongly recommended in that it avoids the need to re-invent every procedure every time it is needed. My only apology here is that limitations of time and experience prevent me from myself offering a more complete library of program routines.

This guide includes the new Delete command and the three new List commands that have been added to the Editor Assembler since the original (Dick Smith) manual was written.

COMPUTER PROGRAMMING LANGUAGES

The programming language installed in the VZ200 and VZ300 computers is BASIC. BASIC is a high level language. This means that it is a language which is reasonably easy for people to use because it uses words and structures that are easily understood. The BASIC programmer actually needs little knowledge of the workings of a computer in order to be able to write useful programs. BASIC however is an interpretive language, which makes it very slow. In many programs this is not even noticed, as the computer using basic is still very much faster than the person using the computer, but as programs become longer and more complicated the delays can become noticeable, particularly in programs using a lot of graphics displays, long calculations, or long searches.

One way to overcome this disadvantage is to use a lower level language. These work more directly with the computer and so are less subject to the delays that can occur in BASIC. They can also provide a greater flexibility in other respects and are much more efficient in the use of memory space than the higher level languages. For these reasons the lower level languages still remain popular with many computer users, though as memory capacity and computer speeds increase they may tend to become less popular than in the past. However while computers like the VZ remain available the lower level languages are sure to remain popular.

The lowest level language is machine code. This is the language that communicates most directly with the computer and is in fact the only language the computer understands, but it has the disadvantage that it is very difficult for people to understand. A machine code is simply a series of numbers which the computer recognises and which can therefore be used to instruct the computer to do certain things. However even one incorrect number in the series will "crash" the program, i.e. cause it to fail to operate properly. Because of the difficulties for humans trying to write machine code the higher level languages were developed, to "translate" something approximating human language into the machine code understood by the computer.

Assembly language is a second level language, i.e. the first level above machine code. It accepts very simple instructions called "mnemonics" and converts them into the machine code required by the computer. This is much easier than using machine code, but not as easy as using a third level language. The programmer using BASIC or another third level language does not have to worry about the detailed steps the microprocessor takes to perform an operation because the interpreter, or the compiler in some other languages, will accept his "general" instruction and supply all the steps needed to tell the computer what to do. The assembler will not do this so the assembly language programmer must tell the computer exactly what he wants it to do, including each and every step of the process, therefore to use it some knowledge of how the computer works is required. However once the necessary knowledge has been gained and one's programming expertise developed assembly language programming offers power and speed far beyond that obtainable with most third level languages, particularly for small computers such as the VZ.

The complexity of a computer is not in what it can do but in how many times and how fast it can do something. For example, a computer cannot multiply or divide, but it can add and subtract, and it can count. Therefore to make it multiply we simply tell it to start at zero and add a number, then add the number again, and again, etc., until it has done so the required number of times. The cumulative total of these additions can then be read out as the result of the multiplication. Division similarly is accomplished by counting the number of successive subtractions required to reach zero. Using BASIC, or any other third level language, the programmer does not have to think about this because the interpreter (or compiler) includes routines which tell the computer how to multiply, divide, etc. when the appropriate instruction is entered, but when using assembler language the programmer must know exactly what the computer can and cannot do and tell it precisely what he wants it to do. However despite the apparent difficulty this is actually made quite easy by the use of the mnemonics, or opcodes, used in the assembly language. The use of these opcodes with the Editor Assembler makes it a fairly easy process to instruct the computer what to do, even though every procedure must be entered in very small steps.

The VZ200 and VZ300 computers use Zilog mnemonics. These are used because the VZ computers use a Zilog Z80 microprocessor. The actual opcodes recognised by the assembler are found in Appendix A of the original Editor Assembly instruction manual (p10) and can also be found in any Z80 manual or reference book. The set *of opcodes recognized by the VZ assembler appears to be standard Z80 but if any* trouble is encountered when using an opcode listed in another source check that is is in fact one that is recognised by the VZ assembler. The pseudo-operation DEFM, for example, is not supported in the VZ assembler.

Assembly language is a compiled language rather than an interpreted language like BASIC. An interpreted language reads and translates the instructions line by line as it runs, which is one reason BASIC is so slow to run compared with other higher level languages such as C-Basic and Pascal, which are compiler languages. A compiler language must be compiled, or assembled, before it can run. When it is compiled, i.e. assembled, it is changed from the language in which it was written, known as the source code, into machine language, or object code. This must be done before the program can be run. Once compiled the program runs very quickly, as it does not have to wait for each line to be translated as do interpretive languages. It is also very efficient in use of memory space at run time as once assembled it can be run without further reference to the source code or the language it was written in, therefore these do not need to be present in the computer at run time.

Because the program written in the higher level language is called the "source code" and the machine language program that results after compilation or assembly is known as the "object code" a program written in Assembly language is a source code, while the program that results after assembly is the object code.

PROCESSOR ARCHITECTURE

Before commencing to write in assembly language is is nesessary to have some understanding of the "architecture" of the microprocessor chip being used, in this case the Z80. Fundamentally the microprocessor consists of several storage areas called "registers" which can be used to store information in a numeric form. If you are entirely unfamiliar with digital electronics it might be useful to glance through an introductory reference now before continuing, as sooner or later you will need some understanding if you continue with assembly programming, but to start with the recognition that the registers are basically storage areas is probably all that is essential.

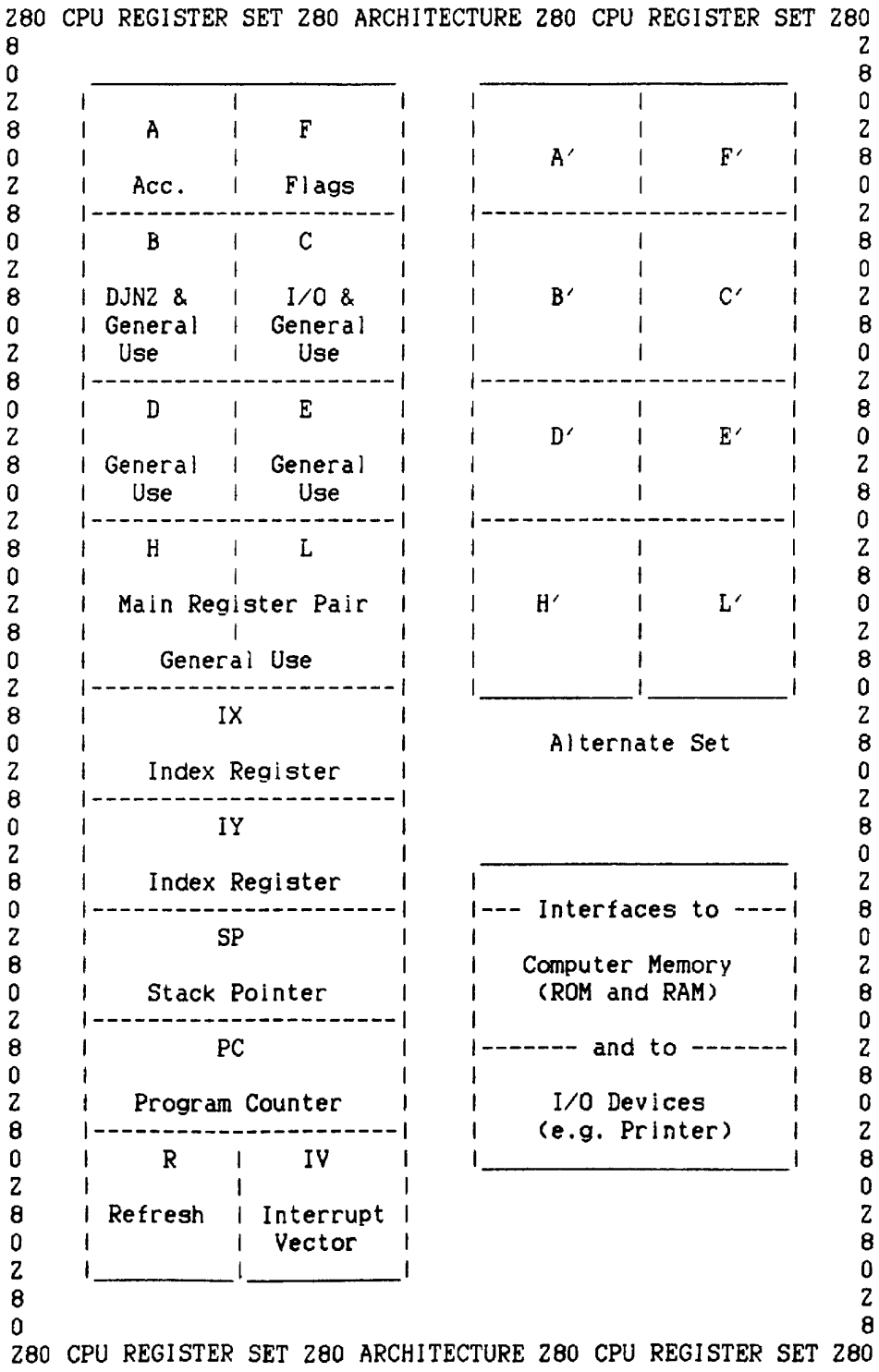
These registers can have "data" put into them or removed, and in the case of the register known as the "accumulator" can have data added to or subtracted from the existing contents. There are several other special purpose registers also, such as the flag and index registers and the stack pointer and program counter, but fundamentally these still are simply storage places for certain information which can then be obtained when required.

Although it could not function, or certainly not very well, without the other registers there is no doubt that the accumulator, the "A" register in the Z80, is the most important. This is because the A register does most of the actual work of the computer, e.g. adding, subtracting, comparing, though some operations can be carried out in some of the other registers when necessary. The Z80 in fact offers several features not always found in "8 bit" microprocessors and because of this is regarded by many people as the "best" all round 8 bit microprocessor, and you should discover as you progress that the flexibility offered by the features available is very useful.

N.B. Registers B & C, D & E, and H & L in the Z80 can be used separately or as pairs, i.e. as two 8 bit registers or as a single 16 bit register. When these registers are used as pairs B and C are the high and low bytes respectively, and similarly with D & E and H & L. H/L = High/Low is a convenient way to remember the byte order.

Until you become very familiar with the Z80 and what it can do you will probably find it useful to keep a copy of the diagram of the register set handy while you work. You should also have handy a list of the opcodes, preferably with explanations. Your first programs will be very fundamental, and probably wont seem to do much, but as you get the feel for what you can do in Assembler language you will begin to realise the power and advantages it offers.

Z80 ARCHITECTURE



Z80 CPU Register Set.

HOW TO USE THE VZ EDITOR ASSEMBLER

The VZ Editor Assembler has two modes, the Command Mode and the Edit mode. The Edit mode is used for the actual writing of programs, i.e. to type programs into the computer from the keyboard. The command mode is used to enter the Edit mode and to carry out other functions such as set origin, assemble, save to tape, etc.

To write a program the Editor Assembler must first be loaded into the VZ in the normal manner, i.e. by use of the CRUN or CLOAD command from the VZ BASIC. (The Editor Assembler auto-executes from both commands). When the assembler has been loaded BASIC is no longer directly accessible as the Editor Assembler now has control of the computer. At the completion of loading the Editor Assembler will be in the Command mode, i.e. programs cannot be written until the Edit mode is entered.

A typical procedure for writing a program is as follows :-

1. Load the Editor Assembler into the VZ using CRUN or CLOAD.

N.B. If you load from the beginning of the tape you will first see a message detailing the new commands that have been included in the Editor Assembler. Allow the tape to keep loading until you see the message identifying the Editor Assembler itself and the COMMAND prompt at the bottom of the screen.

2. When loading is complete the COMMAND prompt should appear. Enter I<Return> (Insert) to begin writing the program. The first unused line number, 001 for a new program, should appear on the screen.

3. Write the program, i.e. type it into the computer via the keyboard. Some people do prefer to write out their programs by hand first but the editing capabilities of the VZ Editor Assembler make this un-necessary, though it is convenient to have some scribble paper handy while working.

> See the "Line Format" section of this (or the original) manual for the correct method of entry and a Z80 reference manual for full details of the Z80 capabilities and opcodes.
>
> If the program is to be tested or otherwise run from the Editor Assembler provision must be made for a return to the Editor Assembler on completion of the Run by including a jump to 31488 (7B00H) as an exit from the program. (See notes under "Set Origin" or "Run" command descriptions).
>
> It is also useful to make line 001 a Comment line, e.g. Program Name, even apart from considerations of good documentation, as it is not possible to insert a line before line 1 and therefore it can be very inconvenient to insert a line at the beginning of a program if it has started with a program line. However because in all but the simplest programs the first several lines should be comment lines for program identification and explanation it is best to establish good programming habits in this respect from the begining anyway.

If a syntax error is made when entering a program line an error message will be displayed and the error must be corrected to proceed. If the line has no syntax errors it will be accepted and the next line number will be displayed.

4. When the program is complete check it and edit the source code to remove any mistakes, e.g. typing errors etc., that remain. [The Editor Assembler only detects "illegal" entries: it can miss many typing errors and cannot detect mistakes in the program sequence itself apart from certain types of mistakes that may become apparent later during the Assembly process.]

5. Set the Origin to the first free byte after the source file using O<Return>. i.e. type the letter O. for Origin, NOT the number zero, then <Return>.

The Origin may be set to a particular memory location for practical programs but beginners will want to test run their programs so should start by setting the Origin to the first free byte after the source file. (Setting the origin to any other memory location could result in a program crash and loss of the program if the location selected is one already in use or allocated for particular use. A good knowledge and understanding of the memory map of the computer is required before the programmer can select the origin for himself).

6. Assemble the program (A<Return>).

7. Re-Edit to remove any errors detected during assembly (the numbers of lines with errors are displayed during assembly) then reset the Origin and re-Assemble the program (as before).

8. Re-Edit etc. again (and again) if necessary until the program assembles without errors. (A printout of the program listing can be done at this stage if required - see Printer Options. N.B. that it can also be convenient to do a printout at earlier stages so as to have a "hard copy" of the program listing to work with when de-bugging a program).

9. Unless the program is very short and simple, i.e. easy to type in again, it should be saved before running (see "Tape Commands", elsewhere in this manual).

10. Finally, when you are sure you have removed all errors. Run the program by typing R<Return> followed by a Y when you are prompted "Are you sure?" (This check is to ensure that the program is not inadvertently Run before it is ready as this could result in loss of the program). If there are no bugs or other wrong commands in the program it should run as intended and then return control to the Editor Assembler. If any errors remain the program may "lock up", i.e. it may not return control to the Editor Assembler. If a program does lock up the only way to unlock it is to switch off and re-start the computer, thus losing all that was in it before the lock up. This is why it is so important to Save every program before a Run is attempted.

If a syntax error is made when entering a program line an error message will be displayed and the error must be corrected to proceed. If the line has no syntax errors it will be accepted and the next line number will be displayed.

4. When the program is complete check it and edit the source code to remove any mistakes, e.g. typing errors etc., that remain. [The Editor Assembler only detects "illegal" entries: it can miss many typing errors and cannot detect mistakes in the program sequence itself apart from certain types of mistakes that may become apparent later during the Assembly process.]

5. Set the Origin to the first free byte after the source file using O<Return>. i.e. type the letter O, for Origin, NOT the number zero, then <Return>.

The Origin may be set to a particular memory location for practical programs but beginners will want to test run their programs so should start by setting the Origin to the first free byte after the source file. (Setting the origin to any other memory location could result in a program crash and loss of the program if the location selected is one already in use or allocated for particular use. A good knowledge and understanding of the memory map of the computer is required before the programmer can select the origin for himself).

6. Assemble the program (A<Return>).

7. Re-Edit to remove any errors detected during assembly (the numbers of lines with errors are displayed during assembly) then reset the Origin and re-Assemble the program (as before).

8. Re-Edit etc. again (and again) if necessary until the program assembles without errors. (A printout of the program listing can be done at this stage if required - see Printer Options. N.B. that it can also be convenient to do a printout at earlier stages so as to have a "hard copy" of the program listing to work with when de-bugging a program).

9. Unless the program is very short and simple, i.e. easy to type in again, it should be saved before running (see "Tape Commands", elsewhere in this manual).

10. Finally, when you are sure you have removed all errors, Run the program by typing R<Return> followed by a Y when you are prompted "Are you sure?" (This check is to ensure that the program is not inadvertently Run before it is ready as this could result in loss of the program). If there are no bugs or other wrong commands in the program it should run as intended and then return control to the Editor Assembler. If any errors remain the program may "lock up", i.e. it may not return control to the Editor Assembler. If a program does lock up the only way to unlock it is to switch off and re-start the computer, thus losing all that was in it before the lock up. This is why it is so important to Save every program before a Run is attempted.

Some Things You Need to Know About This Editor Assembler

The following pages provide information on various things you need to know about this Editor assembler in order to be able to use it effectively.

STATUS LINE (Top Line on Screen)

The top line on the screen is the Status line and is shown in inverse to the rest of the screen. It shows the free memory (FM), the currently set origin (ORG), and the state of the parameters as currently set (SPAR). A number, 1, 2, or 3, indicates the state of parameter N and if A, B, or C is on then this will be shown by the appropriate letter(s) following the number for N.

For example, if free memory equals 10500, the currently set origin is 32000, N = 2, A = on, B = off, and C = on, then the Status line will read :-

```
FM=10500     ORG=32000     SPAR=2A  C
```

Whenever there is a hold on a listing (H), a forced halt in an assembly (:H), or a hold on error status situation then the top line will flash to show you that the program is waiting for you to press the <C> key.

(Explanation of the Parameters can be found under "Assembly and Listing Options" and some explanation of these and of the importance of the Origin can be found under "Commands").

INTERPRETATION OF NUMBERS

The Assembler recognises both decimal and hexadecimal numbers. All Hex. numbers must end with the suffix H. and because all numbers must start with a digit to avoid being interpreted as labels a 0 (zero) must precede any Hex. number that would normally start with a letter.

Examples :-

```
100 = 100 in decimal.
1000H = 1000 in Hex. (equal to 4096 in decimal).
65535 = 65535 in decimal.
0FFFFH = FFFF in Hex. (equal to 65535 in decimal).
```

Note that in the last example the numeral 0 (zero) has to precede the hexadecimal number FFFF because it starts with a letter and therefore would not be recognised by the Editor Assembler as a number without the zero preceeding it. In the second example this was not necessary because that hex. number starts with a numeral.

ARITHMETIC OPERATIONS

The Assembler is capable of performing only addition and subtraction and then only once in any equation.

Examples :-

```
100 + 10
1000H + 0AF0H
TES1 + 1245
234 - 12
10 - 2300
1089H - TES1
TES1 + TES2
```

All of the above are legal with TES1 and TES2 being labels. The following two examples are NOT legal because there is more than one operation in each equation :-

```
120 + 2300 - 12
TES1 + 123 + 12
```

Note that although addition and subtraction are the only arithmetic operations directly available multiplication and division can be performed by using sequential addition or subtraction in combination with the counting capability of the microprocessor and ultimately the mathematical capabilities of the computer are dependent entirely on the programmer's expertise. (See also paragraph 5 of the section on "Computer Programming Languages").

LINE FORMAT

In this assembler provision is made for four types of lines, Program lines, Comment lines, Command lines, and Message lines.

Program Lines

The standard program line format consists of a LABEL, an OPCODE, and an OPERAND. However the label is optional and some opcodes do not require an operand therefore these two positions are not always occupied, e.g. :-

```
001 ABCD    LD      HL,1234H
002         INC     HL
003         NOP
```

This program is not a practical example, as the instructions entered have been chosen simply to show examples of the three alternatives within a program line. Note that although labels and operands are not always needed an opcode is essential in every program line.

In this assembler Comments are not allowed in a normal program line.

If you wish to place a label on a line it must occupy the first 1 to 4 places on the line. It cannot be more than 4 characters long.

No spaces are allowed before a label because if a line begins with a space the assembler will try to handle the next group of characters as if they were an opcode, e.g. 004TES1 not 004 TES1.

The first character of the Label must be alphabetical while the last 3 may be alphanumeric. No special characters are allowed (e.g. !"#$%&).

At least one space must separate the Label and the Opcode.

If no label was used the first character in the line must be a space to prevent the Assembler trying to treat the Opcode as a Label.

The Opcode must have from 1 to 4 characters. The first character must be alphabetical while the last 3 may be alphanumeric. No special characters are allowed (e.g. !"#$%&). The Opcode must also be one of the 71 Opcodes recognised by this Assembler if it is to be accepted at assembly time. These opcodes are explained briefly in this guide and are also listed in Appendix A (page 10) of the original manual.

If an Operand is required then at least one space must be between it and the Opcode.

No check is done on the Operand but the first space found after the commencement of the Operand will be taken as the end of the line.

If the Label or the Opcode does not conform to the 1 to 4 character length and alphabetical or alpha-numerical format then a LINE FORMAT ERROR will be given and you will be asked to re-enter the line.

Comment Lines

Comment lines are used for program documentation, i.e. to include information about the program that is not actually part of it. If you use plenty of Comment lines to explain what your program is actually doing you, and anyone else who uses it, will find it much easier to debug and modify when necessary. A program that is not explained with plenty of comment lines can be extremely difficult to read and modify at a later date.

Many assemblers allow comments on the program lines but this is not supported in the VZ Editor Assembler.

Comments must be on lines by themselves and each Comment line must start with a semi-colon (;). No space (other than that automatically inserted by the Editor) is allowed before the semi-colon. E.g. :-

```
001 :GREAT GAME
002 ;WRITTEN BY G.Bloggs
003 :LAST UPDATE 20/11/86
```

It is good programming practise for the first several lines always to be comment lines (see section on "How to Use the VZ Editor Assembler").

Command Lines

Command lines are used to send special commands to the Assembler during assembly.

A Command must be on a line by itself and the line must start with a colon (:). No space (other than that automatically inserted by the Editor) is allowed before the colon. E.g. :-

```
001 :AC
```

(This particular command will cause all lines to be listed and will send the listing to the printer as well as to the screen).

The commands that can be set by the command lines are described under "Assembly and Listing Options".

Message Lines

Messages are text that is to be printed out by the program. This is normally provided for in Zilog mnemonics by the DEFM Opcode but this is not allowed in the VZ Editor Assembler.

This is because if the DEFM Opcode was used for defining messages they would be limited to 18 characters, the maximum length an operand can be, due to the line entry format of the Editor. While this is long enough for any operand it does limit messages. Messages longer than 18 characters could be defined with two DEFM Opcodes and it would be the same thing as far as the finished assembly was concerned, but to make message handling a little easier a special line indicator is used for messages. The line indicator used for messages in this Assembler is the asterisk (*).

ages must be on lines by themselves and each message line must start with an erisk (*). No space (other than that automatically inserted by the Editor) is lowed before the asterisk.

he maximum message length is 27 characters. If more than 27 characters are equired to complete the message 2 or more message lines will be needed.

The end delimiter for a message is either an asterisk or the first non-space characters working back from the end of the line.

Here are some examples of message lines :-

```
001 *THIS IS A MESSAGE
002 *  THIS IS A MESSAGE  *
003 ** THIS IS A MESSAGE **
```

Line 3 will be assembled * THIS IS A MESSAGE *. Only the first and last asterisks are message delimiters. The others are part of the message.

If a label is required for a message then EQU $ (Opcode Operand) must be used, e.g. :-

```
001 MES1  EQU  $
002 *THIS IS A MESSAGE
003       LD   HL,MES1
```

In this example the register pair HL will be loaded with the address of the memory location that contains the first letter of the message.

ASSEMBLY AND LISTING OPTIONS

Four options can be set for assembly and listing with the Set Parameters command from the Editor Assembler when in Command mode or by the use of the colon (Command line) within a program. A fifth, the Halt option, can be used with the colon (Comand line) during an assembly. The parameters and their meanings are as follows :-

PARAMETER A. If this is set ON then the Assembler will list all lines and their corresponding assembled code. If this is set OFF then only lines with errors will be listed.

PARAMETER B. If this is set ON then the Assembler will halt on an error and wait for you to press the <C> key before continuing.

PARAMETER C. If this is set ON then any lines listed during a List command or an Assemble command will also be sent to the printer as well as to the screen. [Some versions of the EdAssm do not list to the printer as intended but there are several "patches" available to overcome this problem. Your user group would be the best source of information on this].

PARAMETER N. N can be either 1, 2, or 3, and will decide how the Assembler handles the assembly of messages, i.e. :-

N = 1 All messages will be assembled in ASCII format, e.g. 0 = 48, A = 65.

N = 2 All messages will be assembled with bit 6 set off, e.g. 0 = 48, A = 1. This will give white characters on a black background if written directly to the screen.

N = 3 All messages will be assembled with bit 6 set on. e.g. 0 = 112, A = 65. This will give black characters on a white background when written directly to the screen.

NOTE :- If you are going to use VZ200 or VZ300 ROM calls to send characters to the screen then assemble the characters in ASCII format, i.e. set N = 1.

HALT Option, H. This option can be used with the colon (Command line) during assembly. Selection of this option will cause a forced halt in assembly until you press the <C> key.

N.B. A Command line may be used to set parameters or to force a halt, but it may not do both.

Here is an example of a program using Command lines :-

```
001 TEST   LD     HL,7000H
002        LD     DE,7001H
003 :AC
004        LD     BC,1FFH
005        LD     (HL),32
006 :A
007        LDIR
008        XOR    A
009 :H
010        LD     (26624),A
011        JP     7B00H
```

Line 3 will turn parameters A and C on. this means that lines from now on will be listed to the screen (A) and also sent to the printer (C). Line 6 will turn option C off so that lines will be sent to the screen only and no longer to the printer. Line 9 causes a halt in assembly until the <C> key is pressed.

allable in the VZ Editor Assembler.

mand mode several commands are available in the VZ Editor Assembler.
fall into three groups, Text Entry and Display commands, Tape commands, and
al Purpose commands.

Text Entry and Display Commands

Insert, Delete, Edit, List, Find.

These commands are used to enter the Edit mode to write a new program or to read and/or edit an existing program.

**

N.B. You may have trouble getting started if you don't know this.

In normal English language usage it is assumed that "editing" is performed on something previously written but when working with computers this is not necessarily the case. This Editor Assembler, in common with many other computer programs, e.g. word processors, etc., requires the Edit mode both to write new programs and to edit existing ones. Similarly, the "Insert" command does not apply only to inserting new material into an existing file. In the case of this Editor Assembler the Insert command is used to enter the Edit mode regardless of whether or not there is an existing file. Therefore to begin writing a program the command I (for insert) is used to enter the edit mode. This is self-evident to those already familiar with computers but unfortunately is not always so to beginners.

**

For details on the required program line format see the "Line Format" section of this manual.

To exit the edit mode use <Break> (i.e. <Ctrl>- but often referred to as <Ctrl><Break>). This will return you to the command mode.

The normal VZ cursor left and right functions operate in the Edit mode, i.e. :-

<Ctrl>M = Left
<Ctrl>, = Right

The highest line number available in this Editor Assembler is 999.

Insert

I<Retu

Innn<Re

Edit

E<Return

Ennn<Ret

Ennn:mmm

N.
fo

Insert

I<Return>	(i) Enter the edit mode to write new program lines. The first line number should appear on the screen, followed by the cursor. (ii) Insert lines directly after the line pointed to by the Current Line Pointer. This will be the last line listed. i.e. the one visible just above the command line when Command mode is re-entered after inserting or listing. The next line number should appear on the screen, followed by the cursor.
Innn<Return>	Enter the edit mode to write new lines after line number nnn and before any following lines. The next line number. i.e. nnn + 1. should appear on the screen. All following lines will be automatically re-numbered. Use <Break> (i.e. <Ctrl>-) to exit the Insert mode and return to the Command mode.

Edit

E<Return>	Enter the edit mode to change the line currently pointed to by the Current Line Pointer.
Ennn<Return>	Enter the edit mode to change line number nnn.
Ennn:mmm	Enter the edit mode to change lines nnn and mmm and all lines in between. In this mode after you have edited a line press <Return> and the next line to be edited will be displayed.

N.B. In all the Edit modes a <Break> (<Ctrl>-) will end the edit mode for that line and cancel any changes that have been made.

Delete

D<Return> — Delete the program line pointed to by the Current Line Pointer.

N.B. The Current Line Pointer will now point to the line just below the one deleted except in the case of the line deleted being the last line in the source buffer, in which case it will point to the last line remaining, i.e. the one that was previously above the deleted line.

Dnnn<Return — Delete line number nnn.

Dnnn:mmm<Return> — Delete lines nnn and mmm and all lines in between.

D*<Return> — Delete complete text file.

You will be prompted SURE (Y-N)? Press Y to delete the file or N if you do not wish to delete the whole file. This guards against loss of the file because of inadvertent use of this command.

**** WARNING ****

Whenever a line is deleted an automatic line renumber is done, so all the lines after the line deleted will be one number less. Because of this if you wish to delete several lines you should start with the highest number and work down to the smallest number.

E.g. If you wish to delete lines 15, 24, 38, and 67 you should delete 67 first and 15 last. If 15 was deleted first 24, 38, and 67 qould all be renumbered (to 23, 37, and 66) and a D24<Return> would then delete the wrong line, and again renumber all the following lines. Thus all deletions after the first one would affect the wrong lines.

List

L<Retur

Lnnn<Re

Lnnn:-<

L-:nnn<

Lnnn:mmm

L*<Retur

Cursor U
(<Ct

Cursor D
(<Ctrl>

Find

Fstring<

List

L<Return>	List all lines of the source code currently in the buffer.
Lnnn<Return>	List line number nnn.
Lnnn:-<Return>	List from line nnn to the end of the existing source code.
L-:nnn<Return>	List all lines from the begining of the source code up to and including line nnn.
Lnnn:mmm<Return>	List lines nnn and mmm and all lines in between.
L*<Return>	List the last line, i.e. the final program line.
Cursor Up (<Ctrl><.>)	List the previous line.
Cursor Down (<Ctrl><Space>)	List the next line.

In all the multiple line listings the following keys have the following functions :-

<S> = slow listing down.
<F> = speed listing up.
<H> = hold listing.
<C> = continue listing after a hold.
<Break> (i.e. <Ctrl>-) = exit the list mode.

Find

Fstring<Return> — Find a particular word or sequence of characters in the source program.

This command initiates a search through the source file to find the first occurence of the nominated string. It will list the line and then ask NEXT (Y-N)? If you wish to find the next occurence of the same string press Y. If not press N and you will be returned to the command mode.

Using F<Return> with no string will find the first occurence of the string last asked for, starting the search from the line pointed to by the current line pointer.

The string can be a maximum of 8 characters.

Tape Commands

Tape Save, Tape Load, Tape Verify, Tape Merge, Tape Object.

These commands allow assembler source or object programs to be saved to or loaded from tape, i.e. they provide program storage on tape for later use.

Tape Save

TS:name<Return> — Save your source file to tape for later use.

The name can be a max. of 8 characters long and must start with an alphabetical character. Inverse characters are not recommended because of loading difficulties later in some circumstances.

Tape Load

TL<Return> — Load a source file from tape to memory.

No name is allowed as the program will not search for a file but will load the first file it comes to. The name of the file being loaded will be displayed on the bottom line.

Tape Verify

TV<Return> — Verify that a source file has been saved correctly. As in the TL command no name is allowed or searched for. It is used only to verify tapes made with the TS command, not with the TO command.

Tape Merge

TM<Return> — Merge a source file on tape with one that is currently in memory. As in TL and TV no name is allowed or searched for. This command allows a library of useful routines to be built up and then merged into a source file as required.

Tape Object

TO:name<Return> — Write an object tape, i.e. save the assembled (machine language) program.

This command can be used only after the source program has been assembled and provided that no source lines have been edited, deleted, or inserted since assembling. If the source has been modified in any way a REASSEMBLY REQUIRED error message will be displayed.

The program will auto execute when loaded back into the VZ either with the CRUN or CLOAD command.

Special Purpose Commands

Set Origin. Assemble. Run Program. Set Parameters.

Set Origin

Onnn<Return> Set the origin for the assembled program to nnn.

O<Return> Set the origin for the assembled program to the first free byte after the source file.

This command causes the Editor Assembler to set the origin automatically and is therefore ideal for beginners in all respects.

If a program is assembled here then it can be run with the editor assembler still in memory.

Provided your program does not alter any memory below that set by the Origin command then you may jump back to the Editor Assembler at 31488 (7B00H) and your source file will be intact. This is done by having the jump command as the final command in your program, either in the last program line in simple programs or as part of a conditional loop to enable exit from more complex programs.

Assemble

A<Return> Assemble program, i.e. write an object program from the source program.

The program will be assembled starting from the first free byte after the source file but it will be assembled in reference to the value set by the Origin command, which is the address that the program will load and execute at when it is loaded back into the VZ200 or VZ300 as an object tape.

N.B. The Origin should be set before a program is assembled (see Set Origin command).

Run

R<Return> Run the assembled program.

The program must be correctly assembled before it can be Run. To do this two requirements must be met. If any lines have been edited, deleted, or inserted since assembly you will receive a REASSEMBLY REQUIRED error. If the Origin is not set to the first free byte after the source code using the O<Return> command you will get a WRONG ORIGIN FOR RUN error, i.e. the program cannot be run from the Editor Assembler unless the Origin is set to the first free byte after the source code.

Remember that the first free byte after the source code is constantly changing as you enter source lines so if you wish to test run your programs using the Run command use the O<Return> command before every assembly. You will be prompted SURE (Y-N) if everything is correct to run the program. Assuming that you are sure you are ready to run the program you should run it by pressing Y.

****** WARNING ******

Always save your source file before test running a program in memory because often the slightest mistake will cause a program crash which will result in one of two things: either the computer will become locked up and you will have to turn it off and on again to gain control or the computer will reset itself. In either case all that was in memory will have been lost and you will have to load the Editor Assembler again and then your saved source file. If you have not saved the source file you will have to type it all in again.

If you have not made provision in your source code for a return to the Editor Assembler on completion of the program Run the program will probably default to BASIC on conclusion of the run. The Editor Assembler and program will then have to be reloaded (the source code will have to be re-entered from the keyboard if it was not saved). Therefore if the program is to be test Run provision must be made for a return to the Editor Assembler before assembly of the program. Provided your program does not alter any memory below that set by the Origin command this can be done by including an instruction to jump to 31488 (7B00H) as the final command in your program, either in the last program line in simple programs or as part of a conditional loop to enable exit from more complex programs.

Set Parameters

The parameters are used to set certain assembly and listing options i.e. :-

N = Message format.

1 - ASCII
2 - screen characters white on black background.
3 - screen characters black on white background.

A = List all lines (A On) or only lines with errors (A Off).

B = Halt list at each error (B On) or no halt (B Off).

C = List to printer and screen (C On) or screen only (C Off).

N.B. Some versions of the EdAssm do not List to printer as intended but there are several "patches" available to make the printer operate. The best source of information on these would be your user group.

S<Return> Set parameters to default value.

The default values for the parameters are :-

N = 1
A = Off
B = Off

SNABC<Return> Set parameter to the number given (1-3) and turn A, B, and C on.

Not all four parameters need be given when this command is used but any parameter not specified will automatically be reset to its default value.

For full details regarding parameters etc. see section on "Assembly and Listing Options".

SUMMARY of COMMANDS

Edit commands

Insert	I<Return> Innn<Return>
Edit	E<Return> Ennn<Return> Ennn:mmm
Delete	D<Return> Dnnn<Return> Dnnn:mmm<Return> D*<Return>
List	L<Return> Lnnn<Return> Lnnn:-<Return> L-:nnn<Return> Lnnn:mmm<Return> L*<Return> Cursor Up (<Ctrl>.) Cursor Down (<Ctrl><Space>) <S> = slow listing down. <F> = speed listing up. <H> = hold listing. <C> = continue listing after a hold. <Break> (i.e. <Ctrl>-) = exit the list mode.
Find	Fstring<Return>
Exit Edit mode	<Break> (i.e. <Ctrl>-)

Tape commands

Tape Save	TSname<Return>
Tape Load	TL<Return>
Tape Verify	TV<Return>
Tape Merge	TM<Return>
Tape Object	TO<Return>

Special Purpose commands

Set Origin	Onnn<Return> O<Return>
Assemble	A<Return>
Run	R<Return>
Set Parameters	S<Return> SNABC<Return>

OPCODES RECOGNISED BY THIS ASSEMBLER

The following list gives the Opcode and a typical Operand followed by a brief description of the operation performed.

Only those Opcodes are given which are relevant to the VZ Editor Assembler, however this does seem to be a comprehensive list of Z80 Opcodes.

It would be impractical to list every possible Operand therefore representative examples showing the general patterns of the Operands that can be used with each Opcode have been given and it is hoped that these together with the explanations may provide others with an easier introduction to Assembly Language programming than the writer had.

Program Control Operations

CALL	nn	Call address nn, i.e. go to the sub-routine starting at address nn and follow its instructions.
CALL	*,nn	Conditional Call. The call to the sub-routine at address nn will only be performed if the nominated condition '*' is met.
RET		Unconditional Return.
RET	*	Conditional Return, i.e. returns only if the nominated condition '*' is met.
JP	nn	Jump to memory location nn.
JP	(HL)	Jump to the memory location indicated by the contents of the HL memory pair.
JP	*,nn	Jump to the memory location nn if the nominated condition '*' is met.
JP	*,(HL)	Jump to the memory location indicated by the contents of the HL memory pair if the nominated condition '*' is met.
JR	Label	Unconditional Jump to the location indicated by the Label. [Numeric value of relative jumps does not have to be calculated in Assembly Language programming].
JR	*,Label	Conditional Jump to the location indicated by the Label, i.e. jumps to the labelled program line only if the nominated condition '*' is met. [Numeric value of relative jumps does not have to be calculated in Assembly Language programming].

DJNZ Label — Conditional Jump to the location indicated by the Label. First the contents of register B is decremented. If the result is Not Zero a jump to the program line indicated by the Label will take place. If the result is Zero the jump will not occur and the next instruction will be executed. [The numeric value of the jump does not have to be calculated in Assembly Language programming].

RST nn — Restart at page zero location nn.

NOP — No Operation. Only the program counter advances. [Sometimes used to generate short time delays].

HALT — The microprocessor stops operation except for the execution of NOP's to maintain proper memory refresh activity. This is used at the termination of a program or when waiting for an interrupt to occur, i.e. an interrupt is required to restart after a Halt instruction.

Conditions (*) :-

C — Carry.
Do it if Carry flag is Set, i.e. if C = 1

NC — Non-Carry.
Do it if Carry flag is Reset, i.e. if C = 0

M — Sign Negative.
Do it if Sign flag is negative, i.e. if S = 0

P — Sign Positive.
Do it if Sign flag is positive, i.e. if S = 1

PE — Parity Even.
Do it if Parity is Even, i.e. if P/V = 1

PO — Parity Odd.
Do it if Parity is Odd, i.e. if P/V = 0

Z — Zero.
Do it if Zero flag is Set (Zero condition), i.e. if Z = 1

NZ — Non-Zero.
Do it if Zero flag is Reset (Non-Zero condition), i.e. if Z = 0

Data Transfer Operations

Note that the general pattern of the operand is as follows :-

First part, i.e. before comma = Load TO. i.e. the value here will be changed by the operation.

Second part, i.e. after comma = Load FROM, i.e. the value here will not be changed by the operation.

Brackets enclose a memory address or a register whose contents specify a memory address.

LD	A,nn	Load the Accumulator Direct with one byte of data, nn.
LD	BC,nn	Load register pair BC Direct with two bytes of data, nn.
LD	A,B	Load the Accumulator with the contents of register B.
LD	C,A	Load register C with the contents of the Accumulator.
LD	A,(BC)	Load the Accumulator with the contents of the memory location pointed to by register pair BC.
LD	(BC),A	Load the Address pointed to by register pair BC with the contents of the Accumulator.
LD	A,(nn)	Load the Accumulator with the contents of memory location nn.
LD	(nn),A	Load memory location nn with the contents of the Accumulator.
LD	(nn),BC	Load memory locations nn and nn+1 with the contents of register pair BC. The contents of C are loaded into the nominated address and the contents of B into the next address.
IN	A,(n)	Data from the input port 'n' is loaded into the Accumulator.
IN	B,(C)	Data from the input port specified by the contents of register C is loaded into register B.
OUT	n,A	The contents of the Accumulator are output to Port 'n'.
OUT	(C),B	The contents of register B are output to the Port specified by register C.
EX	DE,HL	Exchange the contents of DE and HL registers.
EX	AF,AF'	Exchange the contents of the Accumulator and Flag register with the contents of the Alternate Accumulator and Flag register.

EX	(SP),HL	Exchange the contents of the memory location addressed by the Stack Pointer with the contents of the L register, and the contents of the next address, i.e. SP+1, with the contents of the H register.
EXX		Exchange the contents of the general purpose registers with the contents of the corresponding alternate registers.
PUSH	C	Place the byte from register C onto the stack (at the address of the pointer less one).
PUSH	HL	Place the two bytes from register pair HL onto the stack. The contents of the High Order register are stored in the stack at the address of the pointer less one and the contents of the Low Order register are stored at the address of the stack pointer less two.
POP	C	Remove one byte from the stack and load it into C register.
POP	AF	Remove two bytes from the stack. The first byte is loaded into F and the second into A.

Block Data Transfer & Search Operations

LDD	The contents of the memory location indicated by the contents of the HL register pair is transferred to the location pointed to by the contents of the DE register pair. After the data has been transferred both HL and DE are decremented by one. The "counter-register" pair BC is also decremented by one.
LDDR	This is the same as LDD except that the instruction will be repeated until the value in the "counter-register" pair BC goes to zero.
LDI	This is the same as LDD except that the HL and DE pairs are incremented by a count of one instead of being decremented. The "counter-register" pair BC is still decremented.
LDIR	This is the same as LDI except that the instruction will be repeated until the value in the "counter-register" pair BC goes to zero.
CPD	The contents of the memory location indicated by the HL register pair is subtracted from the accumulator and the result discarded. Both HL and BC are decremented.
CPDR	This is the same as CPD except that it is repeated until either BC = 0 or A = HL.

CPI — The contents of the memory location indicated by the HL register pair is compared with the Accumulator. HL is then incremented and BC decremented.

CPIR — This is the same as CPI except that it is repeated until either BC = 0 or A = HL.

Input/Output Operations

IND — *Input from a port specified by the contents of register C. One* byte of data is transferred to the memory location addressed by the contents of the HL register pair. The values in registers B and HL will be decremented at the end of this instruction.

INDR — This is the same as IND except that it is repeated until register B = 0.

INI — This is the same as IND except that the HL register contents is decremented instead of incremented at the end of the instruction.

INIR — This is the same as INI except that it is repeated until register B = 0.

OUTD — Outputs data from the memory location specified by the contents of the HL register pair to the port specified by the contents of register C. The contents of the B register and of the HL register pair will both be decremented.

OTDR — As for OUTD except that the process is repeated until B = 0.

OUTI — Outputs data from the memory location specified by the contents of the HL register pair to the port specified by the contents of register C. *The contents of the B register will be decremented* but the contents of the HL register pair will be incremented.

OTIR — As for OUTI except that the process is repeated until B = 0.

Arithmetic & Logic Operations

Note that the general pattern of the operand is as follows :-

First part, i.e. before comma = operation TO, i.e. the value here will be changed by the operation.

Second part, i.e. after comma = operation FROM, i.e. the value here will not be changed by the operation.

Brackets enclose a memory address or a register whose contents specify a memory address.

ADD	A,nn	Add the value nn to the Accumulator.
ADD	A,B	Add the value in the B register to the Accumulator.
ADD	HL,BC	Add the value in the BC register pair to the HL register pair.
ADD	IX,BC	Add the value in the BC register pair to the Index register.
ADD	A,(nn)	Add the value of the byte stored at address nn to the A register
ADC	A,nn	Add with carry the value nn to the Accumulator.
ADC	A.B	Add with Carry the value in the B register to the Accumulator.
ADC	HL,BC	Add with Carry the value in the BC register pair to the HL register pair.
ADC	A,(nn)	Add with Carry the value of the byte stored at address nn to the A register.
SUB	nn	Subtract the value nn from the Accumulator.
SUB	B	Subtract the value in the nominated register from the Accumulator.
SUB	(HL)	Subtract the value in the memory location pointed to by the HL register pair from the Accumulator.
SBC	nn	Subtract the value nn and the Carry Flag from the Accumulator (subtract with carry).
SBC	A,B	Subtract the value in the nominated register and the Carry Flag from the Accumulator (subtract with carry).
SBC	A,(HL)	Subtract the value in the memory location pointed to by the HL register pair and the Cary Flag from the Accumulator (subtract with carry).
SCF		Set Carry Flag, i.e. C = 1
CCF		Complement Carry Flag, i.e. reverse (or invert) the condition of the Carry flag.

CPL		Complement Accumulator. i.e. change all 1's to 0's and all 0's to 1's.
NEG		Two's complement the Accumulator, i.e. change all 1's to 0's and all 0's to 1's then add 1 to the result. [i.e. Change the sign of the number in the Accumulator].
CP	A	Compare the Accumulator with itself.
CP	B	Compare the value in the B register with the value in the Accumulator.
CP	(nn)	Compare the value in address nn with the value in the accumulator.
DAA		Decimal Adjust Accumulator. Produces one digit for the four least significant bits and one for the four most significant bits. The carry flag is set to 1 if an overflow occurs.
INC	B	The contents of the nominated register is incremented by 1.
INC	HL	The contents of the nominated register pair is incremented by 1.
INC	(HL)	The contents of the memory location pointed to by the nominated register pair is incremented by 1.
DEC	B	The contents of the nominated register is decremented by 1.
DEC	HL	The contents of the nominated register pair is decremented by 1.
DEC	(HL)	The contents of the memory location pointed to by the nominated register pair is decremented by 1.
AND	A	Logic AND the Accumulator with itself.
AND	B	Logic AND the nominated register with the Accumulator.
AND	(nn)	Logic AND the byte at the address nn with the Accumulator.
AND	nn	Logic AND the data nn with the Accumulator.
OR	A	Logic OR the Accumulator with itself.
OR	B	Logic OR the nominated register with the Accumulator.
OR	(nn)	Logic OR the byte at the address nn with the Accumulator.
OR	nn	Logic OR the data nn with the Accumulator.
XOR	A	Exclusive OR the Accumulator with itself.
XOR	B	Exclusive OR the nominated register with the Accumulator.
XOR	(nn)	Exclusive OR the byte at the address nn with the Accumulator.
XOR	nn	Exclusive OR the data nn with the Accumulator.

Rotate and Shift Operations

RL	n	Rotate operand n Left through Carry (9 bit shift)
RR		Rotate operand n Right through Carry (9 bit shift)
RLA		Rotate Accumulator Left through Carry
RRA		Rotate Accumulator Right through Carry
RLC	r	Rotate register r Left with branch Carry (8 bit shift)
RRC	r	Rotate register r Right with branch Carry (8 bit shift)
RLCA		Rotate Accumulator Left with branch Carry (8 bit shift)
RRCA		Rotate Accumulator Right with branch Carry (8 bit shift)
RLD		Rotate Left Decimal
RRD		Rotate Right Decimal
SLA		Arithmetic Shift Left
SRA		Arithmetic Shift Right
SRL		Logical Shift Right

Bit Operations

BIT	n,A	Bit test. If bit n of the A register is '0' the Zero Flag is set to '1'.
BIT	n,(nn)	Bit test. If bit n of the byte at address nn is '1' the Zero Flag is set to '0'.
SET	n,F	Set Bit 'n' in the specified register to the logic One condition.
SET	n,(F)	Set Bit 'n' in the memory location indicated by the specified register to the logic One condition.
RES	n,(C)	Reset Bit 'n' in the specified register to the logic Zero condition.

Stack and Stack Pointer Operations

LD	SP,rr	Load the Stack Pointer from the nominated register.
EX	(SP),HL	Exchange HL with top of stack, i.e. exchange the contents of the memory location addressed by the Stack Pointer with the contents of the L register, and the contents of the next address, i.e. SP+1, with the contents of the H register.
EX	(SP),IX	Exchange IX with top of stack.
EX	(SP),IY	Exchange IY with top of stack.
PUSH	C	Place the byte from register C onto the stack (at the address of the pointer less one).
PUSH	HL	Place the two bytes from register pair HL onto the stack. The contents of the High Order register are stored in the stack at the address of the pointer less one and the contents of the Low Order register are stored at the address of the stack pointer less two.
PUSH	rr	Push contents of register pair rr onto the stack.
PUSH	IX	Push contents of the IX register onto the stack.
PUSH	IY	Push contents of the IY register onto the stack.
POP	C	Remove one byte from the stack and load it into C register.
POP	AF	Remove two bytes from the stack. The first byte is loaded into F and the second into A.
POP	rr	Pop to register pair rr from the stack.
POP	IX	Pop to register IX from the stack.
POP	IY	Pop to register IY from the stack.

Interrupt and Machine Control Operations

EI		Enable the maskable Interrupt.
IM	n	Set Interrupt Mode to n. (n = 0, 1, or 2).
DI		Disable a maskable interrupt signal.
RST	nn	Restart at page zero location nn.
RETN		Return from a non-maskable interrupt.

RETI		Return from Interrupt.
NOP		No Operation. Only the program counter advances. [Sometimes used to generate short time delays].
HALT		The microprocessor stops operation except for the execution of NOP's to maintain proper memory refresh activity. This is used at the termination of a program or when waiting for an interrupt to occur, i.e. an interrupt is required to restart after a Halt instruction.
LD	I,A	Load the Interrupt vector register with the contents of the Accumulator.
LD	A,I	Load the Accumulator with the contents of the Interrupt vector register.
LD	R,A	Load the Memory Refresh register with the contents of the accumulator.
LD	A,R	Load the Accumulator with the contents of the Memory Refresh register.

Pseudo-Operations

EQU	Equates a label to another label or a numeric value.
DEFB	Defines constants and variables in the program. The argument for DEFB is a numeric or symbolic expression that can be resolved in eight bits.
DEFW	Defines constants and variables in the program. The argument for DEFW is a numeric or symbolic expression that can be resolved in sixteen bits.
DEFS	Reserves a number of bytes in memory without actually filling it with meaningful data, e.g. for allocation of I/O buffers and working storage areas.

N.B. Pseudo-Operations are not Opcodes that are recognised by the microprocessor; they are additional opcodes which give instructions to the assembler.

Summary of Opcodes and Types of Operations.

ADC	Arithmetic & Logic
ADD	Arithmetic & Logic
AND	Arithmetic & Logic
BIT	Bit
CALL	Program Control
CCF	Arithmetic & Logic
CP	Arithmetic & Logic
CPD	Block Data Transfer & Search
CPDR	Block Data Transfer & Search
CPI	Block Data Transfer & Search
CPIR	Block Data Transfer & Search
CPL	Arithmetic & Logic
DAA	Arithmetic & Logic
DEC	Arithmetic & Logic
DEFB	Pseudo-Operation
DEFS	Pseudo-Operation
DEFW	Pseudo-Operation
DI	Interrupt & Machine Control
DJNZ	Program Control
EI	Interrupt & Machine Control
EQU	Pseudo-Operation
EX	Data Transfer
	Stack & Stack Pointer
EXX	Data Transfer
HALT	Program Control
	Interrupt & Machine Control
IM	Interrupt & Machine Control
IN	Data Transfer
INC	Arithmetic & Logic
IND	Input/Output
INDR	Input/Output
INI	Input/Output
INIR	Input/Output
JP	Program Control
JR	Program Control
LD	Data Transfer
	Stack & Stack Pointer
	Interrupt & Machine Control
LDD	Block Data Transfer & Search
LDDR	Block Data Transfer & Search
LDI	Block Data Transfer & Search
LOIR	Block Data Transfer & Search
NEG	Arithmetic & Logic
NOP	Program Control
	Interrupt & Machine Control
OR	Arithmetic & Logic
OTDR	Input/Output
OTIR	Input/Output
OUT	Data Transfer
OUTD	Input/Output
OUTI	Input/Output
POP	Data Transfer
	Stack & Stack Pointer
PUSH	Data Transfer
	Stack & Stack Pointer
RES	Bit
RET	Program Control
RETI	Interrupt & Machine Control
RETN	Interrupt & Machine Control
RL	Rotate & Shift
RLA	Rotate & Shift
RLC	Rotate & Shift
RLCA	Rotate & Shift
RLD	Rotate & Shift
RR	Rotate & Shift
RRA	Rotate & Shift
RRC	Rotate & Shift
RRCA	Rotate & Shift
RRD	Rotate & Shift
RST	Program Control
	Interrupt & Machine Control
SBC	Arithmetic & Logic
SCF	Arithmetic & Logic
SET	Bit
SLA	Rotate & Shift
SRA	Rotate & Shift
SRL	Rotate & Shift
SUB	Arithmetic & Logic
XOR	Arithmetic & Logic

Starter Programs.

First Load the Editor Assembler into VZ. If you load from the beginning of the tape you will first see a message detailing the new commands that have been included in the Editor Assembler. Allow the tape to keep loading until you see the message identifying the Editor Assembler itself and the COMMAND prompt at the bottom of the screen.

Program No. 1.

This program will

1. Clear the screen by calling a sub-routine resident in the VZ 200/300.
2. Provide for a Return to the Editor Assembler after a program Run.
3. Provide a Time Delay for display to the screen so that it remains long enough to be visible.

Enter I<Return>

This is the Insert Command that allows you to enter the Edit mode to write a program. The first line number (001) should now appear on the screen.

Enter :TEST

This is a Comment Line. It is good practice to use Comment lines frequently even though they are not actually used in the program, as a well documented program is much easier to understand when you return to it at a later date. Also, by making the first line a comment line we make it easier to insert new lines ahead of the first actual program line later if we want to. [The Editor Assembler does not allow new lines to be inserted ahead of line 1.]

Enter <Space>CALL<Space>1C9H

This is a Call to the clear screen routine that is resident in the VZ BASIC at address 1C9H, i.e. hexadecimal 1C9 (decimal 441). We are starting with this so that we can have the screen clear to see the results of the programs we write.

Enter <Space>JP<Space>31488

This is a Jump back to the Editor Assembler program to avoid the delay caused by re-loading the program when a Run defaults back to BASIC. The decimal address has been used here, but the hex. address 7B00H will do exactly the same job.

Enter <Ctrl>- [often referred to as <Ctrl><Break>]

This is the VZ BREAK command, but in the Editor Assembler it is used to exit the Edit mode. You should now be back in the Command mode, as indicated by the COMMAND prompt at the bottom of the screen.

The Program as it stands will simply clear the screen and then return to the Editor Assembler. To test this we must first set the Origin to the first free space after the source code, then Assemble the program, and finally Run it. Let's try it.

Enter O<Return>

N.B. This is the letter O for Origin. If you enter zero instead you will be advised that you have entered an ILLEGAL COMMAND. When this happens simply re-enter the correct command at the prompt.

Enter A<Return>

During program assembly you will first see PASS # 1 on assembly of the Opcodes etc., then PASS # 2 when the Jumps are checked. Any errors in your programs will be located during assembly and must be corrected before the program will Run. When the assembler has completed both passes and given a zero error message the program is ready to Run.

Enter R<Return>

You will be prompted SURE (Y-N). As the Origin was set to the first free byte after the source code and the program includes a Jump back to the Editor Assembler we can safely Run it, therefore:-

Enter Y

The screen will be cleared, but the Editor Assembler returns so quickly that you will not have noticed the clearing of the Status Line (Top line in inverse characters across the screen). You will also see COMMAND Y at the bottom of the screen. This is probably there because the program ran so quickly that you still had your finger on the Y key when the Editor Assembler returned after the program run was complete. You can use the normal VZ "Rubout" function to remove the Y or simply press return to get a new COMMAND prompt. (The Y will be reported as an ILLEGAL COMMAND but this doesn't matter as you are automatically prompted for another command).

As you have seen, things happen very quickly in assembly languge programming so before we start doing anything else let's slow it down a little so that we can see what's happening. For this we need a Delay routine. At the moment we only need it once, but a routine like this could be used in several places in a long program so we will start by giving it a Label. This must be from one to four characters long, and to make things easy it should be something that is easily related to the function of the routine, so let's call this one DLY. (If we use other delay routines in the same program we could easily call them DLY1, DLY2, etc.).

The NOP instruction can be used for brief pauses in program execution, as when the computer needs time to sort something out, but such a brief pause as is caused by even a hundred NOP's would never be noticed by a human being so a delay loop must be set up. However not only must it be set up, but it must also have an exit, or the delay will be forever, or until the computer is turned off, whichever comes first. In the Z80 microprocessor the DJNZ function is ideal for setting up time delays. (Ref. description of DJNZ under "Program Control Operations" to see how it works).

Before progressing any further let's see if our original program is still present. (It certainly should be, if it was entered as above).

Enter L<Return>

You should see this :-

```
COMMAND    L
001 :TEST
002        CALL 1C9H
003        JP   31488

COMMAND    (flashing cursor)
```

Now Enter I2<Return>

We could have entered I002 but in practice the non-significant zeros in the line number are not needed, so why bother typing them. We are now Inserting after line 2, before what was originally line 3. Note however that the original line 3 is now line 4 because of the automatic re-numbering carried out by the Editor Assembler each time a new line is inserted. The prompt is now at the new line 003.

Enter <Space>LD<Space>B,0<return> [Enter a zero this time, not a letter]

The assembler should move on to the next line when you press Return.

Enter DLY<Space>DJNZ<Space>DLY<Return>

The Assembler again moves all the following lines down one to enable you to insert another new line, but for the moment let's pause and try the revised program to see what happens at this stage.

Enter <Ctrl>- [i.e. <Break>]

The Break takes us back to the command mode, and as you did not actually enter anything into the last new line offered it is ignored, as you will see if you List your program again (by using L<Return>).

In these new program lines we have set the B register to zero so that when it is decremented the first time it will go to 255. Because the DJNZ checks for condition AFTER the B register is decremented it does not find the Zero condition it needs to allow the next step in the program to be carried out. Instead it loops to the Label DLY, in this case from the end of the DJNZ instruction back to the beginning of it. Because this is a backwards loop it will keep doing this until the B register returns to zero again. Only then will the DJNZ allow the program to move on to the next line.

Now set the Origin and Assemble the program again, and, if it assembles without errors, Run it.

There should still be no noticeable difference. The screen clears, but the Editor Assembler still returns far too quickly for an entirely blank screen to be seen, and our "Y" response still appears after the COMMAND prompt. We have inserted a delay that is long in computer terms, but still very short in human terms so we will have to somehow increase the time delay to make it visible.

If you wish to see more clearly what you are doing List your program again before making the following additions.

Enter I3<Return>

Enter <Space>LD<Space>C,0<Return>

Enter <Ctrl>- [i.e. <Ctrl><Break>]

This new line has loaded a value of zero into the C register to set it also to zero. We then exited the Edit mode because we don't want to insert anything else before the DJNZ instruction at this stage.

Enter L<Return>

Let's List it all again to make sure we get our numbers right. We want to insert some lines after the DJNZ instruction now.

Enter I5<Return>

Enter <Space>DEC<Space>C<Return>

Enter <Space>JR<Space>NZ,DLY<Return>

Enter <Ctrl>-

In the two program lines just entered we are decrementing register C and then causing the program to jump to DLY if it does not find a Zero condition. Only when the Zero condition occurs will the program go on to the next line. The program will keep returning to the DJNZ loop and repeating it until the JR NZ line finds a Zero condition.

Now set the Origin, Assemble, and Run the program again. This time the screen should briefly become completely clear, and the "Y" response to the SURE prompt should no longer appear after the COMMAND prompt when the program returns control of the computer to the Editor Assembler.

Let's now see if we can add another loop to extend the delay time even further. By now you should be getting familiar with the actual entry procedure so I'll just tell you what to do and leave you to work out the actual keyboard entries.

After line 4 Insert a line which will set the D register to zero.

After line ? (the line that now contains the JR NZ,DLY instruction related to the DEC C) Insert two lines that will Decrement register D and cause the program then to loop back to the DLY label until a Zero condition is found. When you have done this List your program again to make sure it really says what you thought it should, and in the right places. If you are satisfied that it is correct then set the Origin, Assemble, and Run the new version of the program. You should find that the screen clears and stays clear for approximately two minutes before control is returned to the Editor Assembler if you have written the program correctly, so wait patiently or take the opportunity of getting a quick drink while nothing seems to be happening.

Now here is a question for you? Why couldn't we have just used two or more DJNZ loops to achieve the extra time delay instead of going to the trouble of setting up other registers to do effectively the same thing?

```
e.g.        LD   C,0
       DLY  DJNZ DLY
            DJNZ DLY
```

** WARNING ** Save the "correct" program to tape first if you are going to try to find the answer to this question by entering and running the example program given above.

If you try this routine instead of using the extra registers and the JR NZ instructions you will find that the screen clears but control is never returned to the Editor Assembler, and the only way to regain control of the computer is to switch it off and start all over again. Why is this?

If you think carefully about what the DJNZ instruction does you should soon recognise the problem. Need a clue? It causes an "infinite loop", but why?

Just to make sure that we are still together, your program now should look like this :-

```
001 :TEST
002       Call 1C9H
003       LD   B,0
004       LD   C,0
005       LD   D,0
006 DLY   DJNZ DLY
007       DEC  C
008       JR   NZ,DLY
009       DEC  D
010       JR   NZ,DLY
011       JP   31488
```

It could be useful to add some labels now to lines 2 and 3 [use the Edit command to do this]. These will make our program easier to follow, but more importantly, the labels can be called from other parts of the program later if required (with some modification to this part to provide for a return to the correct part of the program on completion of the sub-routine called). Thus, although we are not ready to actually use them yet we can start from the begining to form a habit of thinking in terms of program routines and sub-routines.

Remember also that the correct way to make a program easy to follow is by using Comment lines (e.g. line 1) but as these do involve extra typing and these first programs are very simple it has seemed expedient to leave them out for the time being. Labels do make a program easier to followm but this is NOT their purpose. Their purpose is to provide a unique identifier for a particular part of the program so that this part can be found when it is needed, and as often as it is needed.

We will not write the full sub-routines yet, but let's insert labels in lines 2 and 3 to mark the begining of program sections that could be developed into useful sub-routines. On line 2 add the label CLS, for "Clear Screen", and on line 3 add the label DL1 to mark the beginning of the delay subroutine. If you have successfully Edited these lines your program should now look like this:-

```
001 :TEST
002 CLS  Call 1C9H
003 DL1  LD   B,0
004      LD   C,0
005      LD   D,0
006 DLY  DJNZ DLY
007      DEC  C
008      JR   NZ,DLY
009      DEC  D
010      JR   NZ,DLY
011      JP   31488
```

Now let's return to the program itself. We will concede to the experts that there are probably much better and more efficient delay routines, but for beginners this does the trick. The only extra requirement now is to make the delay shorter to avoid unnecessary waiting to return to the Editor Assembler after a program is Run. This can be done by altering the initial value to which one of the registers was set. [More than one of the values could be changed but it is easier to work with one at a time.]

Remembering that setting the register to zero actually gives a count of 256, which gave a total delay in this program of a about two minutes, we can calculate that a count of 2 should give about a one second delay. In practice this is not exactly right, but it does provide a useful rule of thumb. Let's therefore try setting the D register to 30 instead of 0 (i.e. Edit line 5 to read LD D.30), and see what happens. You should find this gives you a delay of about 10-15 seconds. If you find you need more time to see what shows on the screen in the following programs then simply increase the initial value in register D, or if you find you are spending too much time waiting for your program to return to the Editor Assembler then just decrease the value set in D.

DO NOT REMOVE PROGRAM 1 FROM YOUR EDITOR ASSEMBLER!

If you do not intend to continue with the following programs now you may wish to save Program 1 on tape, both for practice in doing this and to avoid having to type it back in next time. The following programs incorporate Program 1 so as to be able to display their results to the screen.

Program No. 2.

This program will write characters to the screen.

To do this we first need to know the screen addresses. These can be found in the computer Technical Manual and also in the "Quick Reference Section" at the back of this guide. The VZ memory map shows that memory locations 7000H to 7800H (28672 to 29183 decimal) are allocated to Video Display Ram. The actual screen addresses however only occupy a small portion of this, and are found on the "Video Display Worksheet" provided in the Technical Manual.

The actual addresses for display access are from 7000H to 71FFH. The top line of the screen is accessed by addresses 7000H to 701FH, the second line by 7020H to 703FH, etc. The four addresses surrounding the centre of the screen are 70EFH, 70F0H, 710FH, and 7110H. The bottom line goes from 71E0H to 71FFH. If you intend doing work which requires accurate screen positioning you should obtain copies of the Video Display Worksheet provided in the Technical Manual or draw up something similar for yourself on graph paper. For our present purposes we really only need to know that anything sent out to an address in the range from 7000H to 71FFH will be printed on the screen by the Video Interface, or Video Display Processor (VDP).

As we intend to write characters to the screen we also need to know the numeric values that correspond to the various characters, bearing in mind that the microprocessor operates entirely with numbers.

The VZ uses standard ASCII values for uppercase characters therefore the numeric vlaues can be read directly from an ASCII table. The characters of most interes to us at present are those in the standard alphabet, and in ASCII the upper cas versions of these are assigned values through from 65, for capital A, to 90, fo capital Z.

N.B. Although the standard ASCII values are used for VZ300 Upper Case character the VZ computers do not strictly follow tha ASCII codes therefore it is better, to avoid confusion, to refer to the ASCII Code Table and Character Code chart provided on pages 203 and 204 of the VZ300 Main Unit Manual (or the equivalent section of the VZ200 manual) than to refer to ASCII tables from other sources. However a listing of the main character codes for the VZ300 is included in the "Quick Reference Section" at the back of this guide for your convenience.

Beginning with the final version of Program 1 (11 lines) Insert the following lines after line 2, i.e. immediately after the Call to the clear screen routine and before the time delay routine :-

```
003     LD   BC,7040H
004     LD   A,65
005     LD   (BC),A
```

The first of the above lines, which should be line 3 in your program, loads the hexadecimal value of the top left-hand corner screen address into the BC register pair. The next line loads the decimal value for the ASCII "A" into register A (the Accumulator). The third line instructs the microprocessor to load the value in the Accumulator into the address pointed to by the register pair BC. Provided the remainder of the program, i.e. from Program 1, is still intact the screen will be cleared before the value for the letter A is loaded to the screen address, and the time delay will hold the resultant display on the screen long enough for it to be seen. (The VDP actually keeps the character on the screen, once written, which is why the B and C registers can be re-used for the time delay which prevents too early a return to the Editor Assembler).

Once you have inserted these lines correctly you can set the Origin, Assemble the program, and if it assembles error free Run it. You should see an A in the top left-hand corner of the screen for whatever length of time you have set up with the time delay.

Once you have the correct display you should use the Edit command to change the values entered into register A and the BC register pair. Remember that the only valid screen addresses, and therefore the only numbers that should be entered into the BC pair to achieve an output to the screen, are between 7000H and 71FFH. If you want alphabetical characters you should put numbers between 65 and 90 (decimal) into register A, but you could try any number between 0 and 255 decimal (00 to FF Hex.) to see what it gives (what is actually displayed will be determined by the VDP; see VZ300 Main Unit Manual p204). Numbers larger than 255 are unable to be accepted by register A because it is only an 8 bit register.

You should now be able to place any of the available characters anywhere you want them to appear on the screen.

> [You should have noticed by now that if the screen position is not occupied by the Status Line (top line) or the COMMAND prompt and cursor the characters actually remain until they are scrolled out by new data being written to the screen, therefore for test programs such as these the delay routine is not really necessary unless the whole screen is required for display. However by starting with this routine installed you have been able to place characters anywhere you liked on the screen without having them overwritten so quickly that they were not seen].

Program No. 3.

Simple Addition and Subtraction.

Now modify your program so that it is as follows :-

```
001 :TEST                          013       ADD  A,L
002 CLS  CALL 1C9H                 014       LD   (BC),A
003      LD   BC,70E8H             015 DL1   LD   B,0
004      LD   H,65                 016       LD   C,0
005      LD   A,H                  017       LD   D,32
006      LD   (BC),A               018 DLY   DJNZ DLY
007      LD   BC,70F0H             019       DEC  C
008      LD   L,66                 020       JR   NZ,DLY
009      LD   A,L                  021       DEC  D
010      LD   (BC),A               022       JR   NZ,DLY
011      LD   BC,70F8H             023       JP   31488
012      LD   A,H
```

Lines 3 to 10 of this program place values in the H and L registers. These values are also put out to the screen as characters, but because it is not possible to load from H or L to the memory address pointed to by BC the A register is used to transfer the data. Lines 12 and 13 add the values from H and L and the result of this addition is sent to the screen as a (VZ character or block-graphics) character. Lines 15 to 23 now contain the Delay routine and the Jump back to the Editor Assembler. Once again the values used may change within the limitations already referred to. Try, for example, loading a value of 150 into both H and L.

Play around with different values in this program then when you are satisfied with the ADD operation Edit line 13 to read :-

```
013       SUB  L
```

With this change to line 13 the ADD has now been changed to SUB (note that the two instructions do not take the same form). Once again set the Origin, Asemble. and Run the program, and try several runs with different values entered into H and L (and different screen positions if you wish).

In these programs addition and subtraction have been used to alter the displayed characters but these operations can be used for arithmetic calculations also. Admittedly the Z80 CANNOT Multiply or Divide, but because it can Add, Subtract, and Count this poses no serious problem. To Multiply the programmer simply uses successive additions for the required number of times and similarly division is achieved by having the microprocessor count the number of successive subtractions required to bring the original value down to zero.

Finally, You've Only Just Begun.

Finally, you have, in fact, only just begun. As you develop (or otherwise acquire) more machine code routines it is suggested that you keep them stored on tape (or disk if you are fortunate enough to have a disk drive) for ready access when you want to use them, and you may find it useful to add listings of your machine code routines to this guide also, for ready reference when required. I would have liked to provide more "graded examples" of machine code routines in this guide but unfortunately other committments prevent me from putting in the time necessary to do this, so it's over to you. However I hope this at least may have given you an easier introduction to VZ Assembler Programming than I had.

```
001 :TRY THIS
002 CLS  CALL 1C9H
003      LD   BC,7107H
004      LD   A,72
005      LD   (BC),A
006      INC  BC
007      LD   A,65
008      LD   (BC),A
009      INC  BC
010      LD   A,80
011      LD   (BC),A
012      INC  BC
013      LD   A,80
014      LD   (BC),A
015      INC  BC
016      LD   A,89
017      LD   (BC),A
018      INC  BC
019      LD   A,96
020      LD   (BC),A
021      INC  BC
022      LD   A,80
023      LD   (BC),A
024      INC  BC
025      LD   A,82
026      LD   (BC),A
027      INC  BC
028      LD   A,79
029      LD   (BC),A
030      INC  BC
031      LD   A,71
032      LD   (BC),A
033      INC  BC
034      LD   A,82
035      LD   (BC),A
036      INC  BC
037      LD   A,65
038      LD   (BC),A
039      INC  BC
040      LD   A,77
041      LD   (BC),A
042      INC  BC
043      LD   A,77
044      LD   (BC),A
045      INC  BC
046      LD   A,73
047      LD   (BC),A
048      INC  BC
049      LD   A,78
050      LD   (BC),A
051      INC  BC
052      LD   A,71
053      LD   (BC),A
054      INC  BC
055      LD   A,97
056      LD   (BC),A
057 DL1  LD   B,0
058      LD   C,0
059      LD   D,8
060 DLY  DJNZ DLY
061      DEC  C
062      JR   NZ,DLY
063      DEC  D
064      JR   NZ,DLY
065      JP   31488
```

Now set Origin to first free byte. Assemble, and Run it.