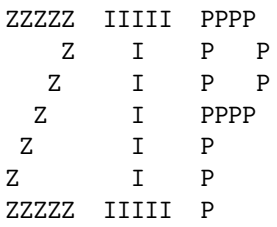

ZIP: Z-language Interpreter Program

Joel M. Berez and Marc S. Blank

December 13, 1982

[Updated by Stu Galley 13 Jan 84, 17 May 84]

Chapter 1

Introduction to ZIP

ZIP is a program running on any of a large variety of machines, which embodies a Z-machine. From the Z point of view, a ZIP may be thought of as providing two functions. It emulates the hardware instructions found on a Z-machine. Also, it provides the software functions of the operating system (ZOUNDS: Z-machine Operating User-Non-Destructive System) ordinarily found on a Z-machine, including program startup and certain service facilities.

This document will describe both functions of ZIP without necessarily differentiating between them. For further information, refer to "ZAP: Z-language Assembly Program," by Joel M. Berez, or to the appropriate not-yet-written document.

ZIP is the lowest level of the ZORK multi-tier game creation and execution system. ZORK is the name of a family of computerized fantasy simulation games. Most of the development system for creating and debugging ZORK games runs on a large mainframe computer in the MDL environment. The final output is a Z program that can run under any ZIP.

ZIP was designed to be usable on any of a large number of small microcomputer systems. The minimum requirements are 40K (possibly 32K) of core with one single-density mini-floppy drive having at least 80K bytes of storage. The design goal also requires no more than a few seconds response time for a typical move.

These goals were achieved by designing a low-level specialized game execution language that could be easily implemented on most microcomputers. To satisfy the core limitation, ZIP pages the disk-resident program. For speed, all modifiable locations are permanently loaded into core along with most tables and some frequently used code. Any extra core available should be used by the ZIP program to buffer disk-resident code as it is used on an LRU or similar basis.

Disk space savings were achieved using an instruction set that is highly space-efficient for ZORK applications. Also, all text is compressed by about one-third.

Chapter 2

ZIP Instruction Format

2.1 General Information

The Z-machine is byte-oriented (assuming 8-bit bytes). Instructions are of variable length and a minimum of one byte.

Data, including instruction operands, are sometimes word-oriented. In this case each word consists of two consecutive bytes (either high-order or low-order byte first, depending upon option), not necessarily beginning on a word-boundary.

Some common examples of word-oriented data are pointers and numbers. Note that although small positive constants can be specified in single-byte format, arithmetic is always done internally with 16-bit words.

Word-boundaries are used in some cases simply to allow pointers in those cases to have twice the addressing range that ordinary byte-pointers would have. Where applicable, these are identified as word-pointers. Note that a word-pointer is a distinct concept from word-oriented data and, in fact, may point to anything.

2.2 Opcode Format

```
Bit #   7 6 5 4 3 2 1 0
2OP     0 m m o o o o o     2-operand (short-form)
1OP     1 0 m m o o o o     1-operand
0OP     1 0 1 1 o o o o     0-operand
EXT     1 1 o o o o o o     extended (0-4 operand)
        (m=mode bits, o=operator bits)
```

The operand format for an instruction depends solely on the opcode format used for the instruction. As can be seen from the above chart, there are only four possibilities.

A given operator will generally use only one of these formats, with the exception that all 2-operand operators may be encoded in either 2OP or EXT format.

Note that the formats were arranged to make decoding easy:

```
        opcode $<$ 128  ==$>$ 2OP
else    opcode $<$ 176  ==$>$ 1OP
else    opcode $<$ 192  ==$>$ 0OP
else                    ==$>$ EXT
```

2.3 Addressing Modes

There are three types of operands: immediate, long immediate, and variable. Operands follow the opcodes in the same order as the mode bits when reading from left to right (high-order to low-order bits).

A long immediate is a 16-bit value that is not further decoded during operand fetching. It may be a twos-complement number, a pointer, or have some other meaning to the operator. An immediate is interpreted exactly as a long immediate with the low-order byte as given and a high-order byte of zero.

A variable operand is a byte that is further decoded as being the identifier of a variable whose value should be used as the actual operand. The number given is interpreted as follows:

```
0       pop a value from the stack
1-15    use local variable #1-15
16-255  use global variable #16-255
```

2.3.1 Single Operand (1OP)

```
Bits:   5 4     Operand
        0 0     long immediate
        0 1     immediate
        1 0     variable
        1 1     undefined
```

2.3.2 Double Operand (2OP)

Bits 6 and 5 refer to the first and second operands, respectively. A zero specifies an immediate operand while a one specifies a variable operand:

```
Bits:   6 5     Operands
        0 0     immediate, immediate
        0 1     immediate, variable
        1 0     variable, immediate
        1 1     variable, variable
```

Note that this format does not allow for long immediate operands. If one is required, the EXT format must be used.

2.3.3 Extended Format (EXT)

In this format there are no mode bits in the opcode itself. All of the mode bits appear in the next byte following the opcode. This mode byte is interpreted as four 2-bit mode-specifiers read from left-to-right as follows:

Bits	1 0	Operand
	0 0	long immediate
	0 1	immediate
	1 0	variable
	1 1	no more operands

Note that extended format does not imply that a given operator takes a variable number of arguments. This format is used in four cases: where a 2-operand operator cannot use 2OP format where an operator requires either three or four operands where an operator is used so seldom that it is undesirable to waste a 2OP, 1OP, or 0OP opcode and, finally, where an operator does indeed take a variable number of operands.

2.4 Instruction Values

Some instructions, such as the arithmetics, return a full word value. These instructions contain an additional byte that specifies to where this value should be returned. This byte is interpreted as a variable in a complementary manner to that described in the previous section.

0	push the value onto the stack
1-15	set local variable #1-15
16-255	set global variable #16-255

2.5 Predicates

Predicate instructions contain an implicit conditional branch instruction. The branch polarity and location are specified in one or two extra bytes in the instruction format. (Note that these bytes would follow the value byte, if any.)

The high-order bit (bit 7) of the first byte specifies the conditional branch polarity. If the bit is on, the branch occurs if the predicate "succeeds." If the bit is off, the branch occurs if the predicate "fails."

The next bit (bit 6) determines the branch offset format. If the bit is on, the offset is the (positive) value of the next 6 bits. If the bit is off, the offset is a 14-bit twos-complement number, where the next 6-bits are the high-order bits and another byte follows with the 8 low-order bits. (Note that these are two consecutive bytes and not a word. Therefore byte-swapping would have no effect.)

If the branch should not occur, execution continues at the next sequential instruction. Otherwise, if the offset is zero, an RFALSE instruction is executed. If the offset is one, an RTRUE instruction is executed. For any other offset, a JUMP is done to the location of the next sequential instruction plus the offset minus two.

Chapter 3

ZIP Instruction Set

3.1 Instruction Metasyntax

Instructions will be individually described in the following format. A heading will show the instruction name followed by its arguments (operands). The heading line is followed by explanatory text.

On the right side of the heading line the valid opcode format(s) is shown followed by the base opcode value (assuming mode bits are all zero). It is implicitly understood that for each 2OP format, there is also a legal EXT format with a base opcode 192 higher.

The opcode format information is optionally followed by /VAL and/or /PRED according to whether the instruction returns a value or is a predicate.

The operands on the heading line are given names indicative of their use:

```
int     twos-complement integer, used arithmetically
word    word of bits for logical operations
any     no special meaning attached
obj     object number
flag    flag number
prop    property number
table   pointer to a table
item    element position in a table
var     number of a variable
str     pointer to a string
fcn     pointer to a function
loc     pointer to a program location
```

3.2 Arithmetic Operations

Any arithmetic operation that returns a value that does not fit in a 16-bit word is in error.

`ADD  int1,int2  2OP:20/VAL`

Adds the integers.

`SUB  int1,int2  2OP:21/VAL`

Subtracts int2 from int1.

```
MUL     int1,int2       2OP:22/VAL
```

Multiplies the integers.

```
DIV     int1,int2       2OP:23/VAL
```

Divides int1 by int2, returning the truncated quotient.

```
MOD     int1,int2       2OP:24/VAL
```

Divides int1 by int2, returning the remainder.

```
RANDOM  int     EXT:231/VAL
```

Returns a random value between one and int, inclusive.

```
LESS?   int1,int2       2OP:2/PRED
```

Is int1 less than int2?

```
GRTR?   int1,int2       2OP:3/PRED
```

Is int1 greater than int2?

3.3 Logical Operations

```
BTST    word1,word2     2OP:7/PRED
```

Is every bit that is on in word2 also on in word1?

```
BOR     word1,word2     2OP:8/VAL
```

Bitwise logical or.

```
BCOM    word    1OP:143/VAL
```

Bitwise logical complement.

```
BAND    word1,word2     2OP:9/VAL
```

Bitwise logical and.

3.4 General Predicates

```
EQUAL?  any1,any2[,any3][,any4] 2OP:1,EXT:193/PRED
```

Is any1 equal to any2, any3, or any4? Note that this instruction differs from the usual 2OP/EXT format in that in the extended form, EQUAL? can take more than two operands. The motivation here was to provide a short (2OP) form for the most common use of this instruction, which would otherwise use EXT format.

```
ZERO?   any     1OP:128/PRED
```

Is any equal to zero?

3.5 Object Operations

Objects have six pieces of information associated with them that may be accessed using the following commands. The object itself is referenced by a one-byte number. Object number zero is a special-case pseudo-object used where an object-pointer slot is empty.

Each object contains 32 1-bit flags, arranged as two words, and numbered from left to right, 0 to 31 (not the usual numbering scheme in this document). There is also a string of text, which is the short description referenced by PRINTD.

Three slots in an object contain pointers to other objects. These pointers are used to link objects together in a hierarchical structure. The LOC slot points to the object that this object is contained in. All objects contained in a particular object are chained together in an arbitrary order via the NEXT slot. The FIRST slot points to one of the objects that this object contains, which is the first object in the NEXT chain.

```
MOVE    obj1,obj2       2OP:14
```

Put obj1 into obj2.

```
REMOVE  obj     1OP:137
```

MOVEs obj to pseudo-object zero.

```
FSET?   obj,flag        2OP:10/PRED
```

Is this flag number set in obj?

```
FSET    obj,flag        2OP:11
```

Set flag in obj.

```
FCLEAR  obj,flag        2OP:12
```

Clear flag in obj.

```
LOC     obj     1OP:131/VAL
```

Return container of obj, zero if none.

```
FIRST?  obj     1OP:130/VAL/PRED
```

Return "first" slot of obj. Fails if none (equals zero) and returns zero.

```
NEXT?   obj     1OP:129/VAL/PRED
```

Returns "next" slot of obj. Fails if none (equals zero) and returns zero.

```
IN?     obj1,obj2       2OP:6/PRED
```

Is obj1 contained in obj2?

```
GETP    obj,prop        2OP:17/VAL
```

Returns specified property of obj. If obj has no property prop, returns prop'th element of default property table.

```
PUTP    obj,prop,any    EXT:227
```

Changes value of obj's property prop to any. Error if obj does not have that property.

```
NEXTP   obj,prop        2OP:19/VAL
```

Returns the number of the property following prop in obj. Error if no property prop exists in obj. Returns zero if prop is last property. Given prop equal to zero, returns first property (i.e. is circular).

3.6 Table Operations

Tables are in fact only a useful logical concept and have no physical form in the Z-machine. (However the assembler, ZAP, does "know" about tables.) Table pointers are simply byte-pointers to appropriate locations in the Z program. Since ZIP assumes nothing about tables, these pointers may be arithmetically manipulated or even randomly generated (if the programmer finds that useful). Note that manipulating arbitrary program locations constitutes "taking the back off" and voids the warranty.

```
GET     table,item      2OP:15/VAL
```

Interpreting the table pointed to as a vector of words, returns the item'th element. In other words, returns the word pointed to by item times two plus table. (Tables begin with element zero.)

```
GETB    table,item      2OP:16/VAL
```

Similar to GET, but assumes a byte table. Returns the byte (converted to a word, of course) pointed to by item plus table.

```
PUT     table,item,any  EXT:225
```

Inverse of GET. Sets the word pointed to to any.

```
PUTB    table,item,any  EXT:226
```

PUTB is to GETB as PUT is to GET. Uses only the low-order byte of any. Error if the high-order byte is non-zero.

```
GETPT   obj,prop        2OP:18/VAL
```

Gets property table prop from obj. Where GETP can only be used with single byte or single word properties, GETPT can be used with properties of any length. It returns a pointer to the property value that may then be used as a table pointer in any other table operation.

```
PTSIZE  table   1OP:132/VAL
```

Given a property table pointer as may be obtained from GETPT, returns the length of this "table" in bytes. Guaranteed to return a meaningless value if given any other kind of table. (Assumes that the byte preceding the table is a property identifier.)

3.7 Variable Operations

A variable, as used in the following instructions, differs from a variable used as an operand. The latter is evaluated to get the actual value of the operand. In contrast, these variables are identified by the already evaluated operands. This allows for the possibility, for example, that one variable may "point" to another variable to be used.

These variable identifiers are interpreted almost as variables are during operand decoding except in regards to the stack, where no pushing or popping occurs:

```
0       use the current top-of-stack slot
1-15    use local variable #1-15
16-255  use global variable #16-255
```

```
VALUE   var     1OP:142/VAL
```

Returns the value of var.

```
SET     var,any 2OP:13
```

Sets the specified variable to any.

```
PUSH    any     EXT:232
```

Pushes any onto the stack.

```
POP     var     EXT:233
```

Pops the top word off the stack and puts it into var. Note that "POP 'STACK" will have the effect of flushing the next to the top word of the stack.

```
INC     var     1OP:133
```

Increments the value of var by one.

```
DEC     var     1OP:134
```

Decrements the value of var by one.

```
IGRTR?  var,int 2OP:5/PRED
```

Increments the value of var by one and succeeds if the new value is greater than int.

```
DLESS?  var,int 2OP:4/PRED
```

Decrements the value of var by one and succeeds if the new value is less than int.

3.8 I/O Operations

Due to the diversity of machines that ZIP may run on, the I/O operations have been designed for the simplest terminal handling features expected to be available. A printing terminal or simple display terminal is required supporting only the basic ASCII control functions. Lowercase capability is useful but unnecessary as long as ZIP does the proper conversion. Line-at-a-time input and output is sufficient for all I/O operations.

Because line lengths may vary, it is up to the particular implementation of ZIP to insure that the line length is not exceeded on output. In general a Z-language program will only output a newline character in cases where a line must be terminated. Most text strings will contain only spaces.

ZIP maintains a line-length output buffer. Printing occurs only when a newline character is output by the program or when the line is filled. In the latter case, the line is broken at the last space, with the remainder being moved to the beginning of the next line. The buffer is also emptied before each READ operation (without going to the next line, if possible).

```
READ    table1,table2   EXT:228
```

Reads and parses a line of input. Table1 is the buffer used to store the characters read. The first byte (read-only) of this table contains the length of the rest of the buffer where the input string is stored. All uppercase characters must be converted to lowercase before READ is finished. This enables the program to reprint words from the buffer without being concerned about case.

Table2 is used to store results of the parse. The first byte (read-only) of this table specifies the maximum number of words (of text, not machine words) that may be stored here. The second byte is used by READ to report the number of words actually read. The rest of the table consists of fixed-length word entries.

READ will fill each entry with three items. First is a 16-bit byte-pointer to the word entry in the vocabulary table, zero if not found. Next is a byte giving the word length as typed (number of ASCII characters). Last is a byte giving the byte-offset of the beginning of the word in the buffer table. (Because of the length byte, the first character in the buffer is at offset 1.) These last two values are used by the program in conjunction with PRINTC to reprint words.

READ reads text until the buffer is full or until it encounters a newline character. Words may be separated by standard break characters (space, tab, etc.) or by self-inserting break characters (usually comma, period, etc.). The self-inserting characters for a given program are specified in the vocabulary table (Chapter 4). Each of these characters not only separates words but is also considered a word itself and may be found in the vocabulary word list.

When parsing a word, it must first be converted to Z string format (Chapter 4) after case conversion, if any. It should be truncated to 6 (5-bit) bytes to fit into two machine words to match the vocabulary table entries. (Note that as in all Z strings, the low-order bit of the last (second) word will be on.) This may actually correspond to less than 6 ASCII characters. If the encoded word is less than 6 bytes, it should be padded with the pad character (5). The words in the vocabulary table are sorted to facilitate a binary search.

Before doing any of the above, READ must empty the output buffer. If a status line exists, it should be updated (see Chapter 4).

```
USL             0OP:188
```

Updates the status line now instead of waiting for the next READ.

```
PRINTC  int     EXT:229
```

Prints the character whose ASCII value is int.

```
PRINTN  int     EXT:230
```

Prints int as a signed number.

```
PRINT   str     1OP:141
```

Prints the string pointed to by str times two. The multiplication is necessary because str in this instruction is a word-pointer, guaranteed to point to a string that has been word-aligned.

```
PRINTB  str     1OP:135
```

Like PRINT, but str here is an ordinary byte-pointer.

```
PRINTD  obj     1OP:138
```

Prints the short description of obj.

```
PRINTI  (str)   0OP:178
```

Prints an immediate string. Interpreted as a 0-operation instruction but immediately followed by a standard string (as opposed to a string-pointer).

```
PRINTR  (str)   0OP:179
```

Like PRINTI but executes a CRLF followed by an RTRUE after printing the string.

```
CRLF            0OP:187
```

Prints an end-of-line sequence (carriage-return/line-feed in ASCII).

```
SPLIT   int     EXT:234
```

If option bit 5 in the mode byte is zero, this operation is ignored otherwise it divides the screen into two windows: #1 occupies int lines, preferably at the top of the screen, and #0 occupies the remainder of the screen. If int is zero, this operation restores the normal screen format. Window #1 is special in that it never scrolls if the program outputs characters beyond the right-hand margin, they are not displayed. [SWG 1/13/84]

```
SCREEN  int     EXT:235
```

If option bit 5 in the mode byte is zero, this operation is ignored otherwise it causes subsequent screen output to fall into window #int. If int is 1, the output cursor is moved to the upper left-hand corner

if int is 0, the output cursor is restored to its previous position. This operation is ignored if the screen is not split, or if int is not zero or one. [SWG 1/13/84]

3.9 Control Operations

`CALL  fcn[,any1][,any2][,any3]  EXT:224/VAL`

Begins execution of the function (see Chapter 4) pointed to by fcn times two, supplying it with any arguments given in the CALL instruction. Note that fcn is a word-pointer and functions are always word-aligned. See RETURN for the method of returning from this instruction.

If fcn equals zero, the CALL is special. In this case, it ignores its other arguments (except for the value specifier) and acts as if it had called a function that did an immediate RFALSE.

`RETURN  any  1OP:139`

Causes the most recently executed CALL to return any and continues execution at the next sequential instruction after that CALL.

`RTRUE  0OP:176`

Does a "RETURN 1," where 1 is commonly interpreted by Z programs as "true."

`RFALSE  0OP:177`

Does a "RETURN 0," where 0 is commonly interpreted by Z programs as "false."

`JUMP  loc  1OP:140`

An unconditional relative branch to the location of the next sequential instruction plus loc minus two (for compatibility with predicates). Note that unlike the predicate argument, this is a full twos-complement word.

`RSTACK  0OP:184`

Does a "RETURN STACK," thereby returning from a CALL and taking the value from the (old) top of the stack.

`FSTACK  0OP:185`

Flushes the top value off the stack.

`NOOP  0OP:180`

No operation, equivalent to a "JUMP 2."

3.10 Game Commands

`SAVE  0OP:181/PRED`

Writes the "impure" part of the game to disk in some recoverable format. The seed for RANDOM should not be saved or restored so that multiple RESTOREs from the same SAVEd game will not necessarily lead to the same results. Other details of the user interface are left to the discretion of the implementor. Note that this instruction is a predicate.

`RESTORE` `OOP:182/PRED`

Recovers a previously SAVEd game and continues execution after the SAVE. If the RESTORE fails, execution should continue (if possible) after the RESTORE in the original game with the instruction failing.

`VERIFY` `OOP:189/PRED`

Verifies the correctness of the game program stored on disk by comparing the 16-bit sum of the bytes in the program, from byte 64 to byte PLENTH*2-1, with PCHKSM. Note that for the preloaded area, the unmodified pages on the disk should be used rather than the pages in core.

`RESTART` `OOP:183`

Reinitializes the game and generally acts as if it had just been started.

`QUIT` `OOP:186`

The game should die peacefully.

Chapter 4

ZIP Data Structure

4.1 Program Structure

A Z-language program begins with the following words:

```
ZVERSION        version of Z-machine used
ZORKID          unique game identifier
ENDLOD          beginning of non-preloaded code
START           location where execution begins
VOCAB           points to vocabulary table
OBJECT          points to object table
GLOBALS         points to global variable table
PURBOT          beginning of pure code
FLAGS           16 user-settable flags
SERIAL          serial number - 6 bytes
FWORDS          points to fwords table
PLENTH          length of program (in words)
PCHKSM          checksum of all bytes
(17 reserved words)
```

ZVERSION is interpreted as two bytes. Regardless of the state of the byte-swap mode, the version byte is always first followed by the mode byte. The Z-machine version described in this document is 3 [SWG 1/13/84], which is the content of the version byte. The mode byte contains eight option bits as follows:

```
Bit #        Interpretation
7-6     reserved
5       splittability:  SWG 1/13/84
            0 means SPLIT \& SCREEN operations are ignored
            1 means they are functional
4       status line existence:
            0 means a status line is present
            1 means there is no status line
3       Tandy bit:
            0 non-Tandy machine
```

```
                1 a Tandy machine is being used
2       machine-specific function:
                - always set to 0 by Z development system
                - available for use by particular interpreters
1       status line format:
                0 means numbers are score and moves
                1 means numbers are hours and minutes
0       byte-swap:
                0 means words are high-order byte first
                1 means words are low-order byte first
```

Note that bits 0 and 1 are set by the ZAP assembler. The others are set by either a loader for a particular machine or the interpreter at start-up time.

ZORKID identifies the game type and its version number. This is checked by RESTORE.

ENDLOD is a particularly significant pointer. A typical Z-machine has a limited amount of primary memory available. Therefore programs are arranged so that most data/code can remain on disk during execution. All locations below ENDLOD must be preloaded in core. These include all modifiable locations in the program. (Attempts to modify other locations should cause an error.) If more memory is available, any or all of the rest of the program may be preloaded.

Due to restrictions on the number of bits available in pointers, the maximum size of a program is 128k bytes. All modifiable data, including anything that a byte-pointer might point to, will be below 64k in this address space. All major tables (VOCAB, OBJECT, etc.) are guaranteed to be below ENDLOD.

The FLAGS word is used to hold user-settable flags that control various interpreter options:

Bit #	Interpretation
15-2	reserved
1	fixed-width font needed
0	script bit

Bit #1 should be checked by every "printing" operation before actually doing any output. If it is on, the output must appear in a type face with all characters the same width, since the game is making a crude picture with the characters. [SWG 5/17/84]

Scripting is an optional feature of the interpreter. When the feature exists and the script bit is set, all terminal output should be echoed to the printer.

The serial number is a six-character ASCII string uniquely identifying each distributed copy of a game. This string will be inserted when each distribution disk is created and will be read by the game program when executed.

PLENTH and PCHKSM are both used by the VERIFY operation. PCHKSM is the 16-bit sum of all bytes from 64 (decimal) to PLENTH*2-1.

4.2 Global Table

This table contains a one-word slot for each global that will be used by the program with its starting value. Note that the first slot (pointed to by GLOBALS) corresponds to variable number 16.

Some interpreters implement a status line, which is a reserved line on the screen that constantly displays status information about the game (updated before each READ or at each

USL). To provide the required information, the first three globals are predefined. Global 16 contains the object number of the current room, which can be used with PRINTD to get its short description. In a score-oriented game (see ZVERSION mode-byte), global 17 contains the number of moves that have been made in the game and global 18 contains the current score. In a time-oriented game, they are minutes and hours, respectively. These two numbers and the string may be printed in any convenient order along with any other desired information.

4.3 Object Table

The first 31 words of the object table form the default property table. This contains values that will be returned by GETP when the corresponding property numbers (1 through 31) are not found in a specified object.

The rest of the table contains the objects themselves, numbered sequentially from 1 to the total number of objects (must be less than 256). An object is formatted as follows:

byte	value
0-1	first flag word, flags 0-15
2-3	second flag word, flags 16-31
4	LOC slot
5	NEXT slot
6	FIRST slot
7-8	property table pointer

The property table pointer points to another table associated with this object:

```
number of words in short description (1 byte)
short description string
property identifier (1 byte)
property value (1-8 bytes)
.
.
.
property identifier
property value
0
```

There may be from 0 to 31 property pairs. Each property identifier has the property number in the low-order 5 bits. The high-order 3 bits are the number of bytes in the property value minus one. Therefore a value may have from one to eight bytes. For searching efficiency, the properties are sorted in inverse order by property number.

4.4 Vocabulary Table

This table contains the words that will be understood by READ, other information for READ, and, optionally, some user-defined information ignored by ZIP:

```
number of self-inserting break characters (1 byte)
character #1 (1 ASCII byte)
.
```

```
.
.
character #n
number of bytes in each entry (1 byte)
number of entries (words) in vocabulary
word #1 (4-byte string)
extra entry bytes for word #1
.
.
.
word #m
extra entry bytes for word #m
```

Words are truncated or padded to cause them to fit into 4 bytes. READ performs the same function, so comparisons work. Words in the vocabulary table are sorted according to this 4-byte value.

4.5 String Format

For maximum storage efficiency, text is encoded in 5-bit byte strings. Characters are packed into 16-bit words from left-to-right (high-to-low), skipping the high-order bit. The last word in each string has the high-order bit set, which is otherwise clear. If the last word is not filled, it is padded with the standard pad character (5), which conveniently is interpreted as a no-op during printing.

The 5-bit code actually encompasses three different character sets: 0, 1, and 2. At any instant during string interpretation (printing) there is a particular permanent mode. A temporary mode can also exist for one character at a time. Each character read is interpreted in terms of the temporary character set if there is one, and otherwise the permanent character set.

The first 6 values are universal over all character sets. 0 means space. 1, 2, or 3 means to use one of the special words (see below). 4 and 5 are shift characters. Each permanently or temporarily changes the character set to one of the other two:

	New Character Set (P=perm, T=temp)		
Old C.S.		4	5
0	1T	2T	
1	1P	0P	
2	0P	2P	

In character set 0, 6 through 31 represent a through z. In character set 1, they represent A through Z. In character set 2, 6 means that the ASCII value specified by the following two bytes, high-order byte first, should be used. 7 represents a new-line character (carriage-return line-feed combination in ASCII). 8 through 31 represent 0 through 9, period, comma, !, ?, _, #, ', ", /, \, -, :, (, and).

At the beginning of each string, the initial permanent character set should be 0, with no temporary mode selected. The encoding algorithm used to create the string also specifies that whenever the current character to be encoded is not in the current permanent character set, the following character is examined. If there is a following character (i.e. not at end of string) and that character is in the same set as the current one, a permanent shift is used. Otherwise a temporary shift is used.

4.5.1 Fwords Table

The fwords table, pointed to by FWORDS and below PRELOD, contains 96 word-pointers to ordinary strings. These strings represent frequently used substrings (usually words) within other strings. Whenever a 1, 2, or 3 byte is encountered in a string that it is being decoded, the following byte is used as a word-offset into the fwords table to select one of the string pointers. The first, second, or third group of 32 words in the table is used, according to whether the initial byte was 1, 2, or 3, respectively. The string interpreter routine is recursively called to handle this new string. When done it returns to continue handling the original string.

Note that the substring is treated as a complete self-contained string. This means that it starts in permanent character set 0, with no temporary set. In the original string, the permanent set is retained across the call to the substring. (Of course, there will be no temporary character set to remember.) The substrings in the fwords table are guaranteed not to contain fwords themselves. Therefore, the string interpreter routine need not necessarily be totally reentrant.

4.6 Functions

A function is a subroutine that is accessed via the CALL and RETURN mechanism. It may optionally have up to 15 local variables, up to 3 of which may be set by the CALL instruction.

A function may be preloaded or disk-resident (or both). It begins on a word-boundary. The first byte specifies the number of local variables to be used by the function (0 to 15). This is followed by one word for each such variable giving its initial or default value. The first one, two, or three variables may be initialized to values supplied in the CALL instruction instead of these. Note that this format allows for optional arguments.

The value words are followed by the first instruction to be executed when the function is called. Execution will continue from that point until a RETURN is executed.