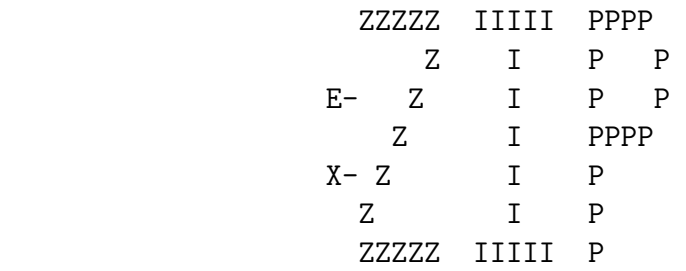

ZIP: Z-language Interpreter Program

Joel M. Berez (ZIP)
Marc S. Blank (ZIP, EZIP)
P. David Lebling (XZIP)

ZIP: December 13, 1982
EZIP: October 26, 1984
XZIP: May 27, 1986

Acknowledgments

ZIP has been worked on by many people. It was first designed by Marc Blank and Joel Berez, and substantially expanded by Blank to create EZIP. XZIP was designed by Dave Lebling and Brian Moriarty, with substantial contributions from Duncan Blanchard and Linde Dynneson.

[pdl 6/3/87]

Contents

Chapter 1

Introduction to ZIP

ZIP is a program, running on any of a large variety of machines, which emulates the non-existent 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 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 Infocom's multi-tier interactive fiction creation and execution system. Most of the development system for creating and debugging these products runs on a powerful computer in the MDL environment. The final output is a Z program that can run under any ZIP.

There are currently three versions of ZIP used in developing games. These are known as ZIP, EZIP, and XZIP. It is intended that ultimately XZIP will replace EZIP. Eventually, as interest in the 8-bit microcomputers diminishes, ZIP will be retired. Finally, there is a special version of EZIP called LZIP ("lower-case EZIP") which is EZIP with game size limitations for certain medium-sized micros.

This document is primarily concerned with EZIP and XZIP. The major differences between these and "classic" ZIP will only be summarized.

EZIP/XZIP was designed to be usable on any of a large number of medium to large microcomputer systems. The minimum requirements are 128K of primary memory with one disk drive having at least 140K bytes of storage. The design goal also requires no more than a few seconds response time for a typical move.

These goals are achieved by designing a low-level specialized game execution language that can be easily implemented on most microcomputers. To satisfy the RAM limitation, ZIP pages the disk-resident program. For speed, all modifiable locations are permanently loaded into RAM along with most tables and some frequently used code. Any extra RAM 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 interactive fiction. Also, all text is compressed by nearly one-half.

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, 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.

Quad-byte-boundaries are used in some cases to allow pointers to have four times the addressing range that ordinary byte-pointers would have. Where applicable, these are identified as quad-pointers.

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 operands)

        (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.

In XZIP, the operand space has been expanded for a further 256 opcodes by means of the EXTOP instruction. EXTOP causes the next byte to be treated as an opcode 256 higher. All such opcodes are decoded as EXTs.

Note that the formats were arranged to make decoding easy:

```
if     (opcode < 128) then 2OP
elseif (opcode < 176) then 1OP
elseif (opcode < 192) then 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:

var	interpretation
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. In the special case of the XCALL instruction (and IXCALL in XZIP), there are two of these mode bytes following the opcode. A 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.

The EXTOP instruction provides for 256 additional EXT opcodes. This opcode (190) tells the interpreter that the next byte is to be treated as an opcode whose value is itself plus 256. EXTOPs are produced by ZAP, so they are usually not seen unless code is being examined byte-by-byte.

2.4 Instruction Values

Some instructions, such as the arithmetics, return a full word value. These instructions contain an additional byte that specifies where this value should be returned. This byte

is interpreted as a variable in a complementary manner to that described in the previous section.

var	interpretation
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.)

If the branch does 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). If the format of the instruction is different in EZIP or XZIP, there may be more than one line of heading. 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.

If the opcode is XZIP only, or differs in its interpretation between XZIP and EZIP, there will be an X (for XZIP) at the right edge of the heading line.

The operands on the heading line are given names and types 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 (quad)
fcn     pointer to a function (quad)
loc     pointer to a program location
```

Optional arguments are indicated by underlining or italics, thus.

The opcode argument information is optionally followed by ¿VAL and/or /PRED according

to whether the instruction returns a value or is a predicate. (This is same format ZAP uses).

3.2 Extended Opcodes

`EXTOP opcode:int` `0OP:190 X`

Tells the interpreter that the following opcode is an extended opcode, meaning that the next byte is an opcode from a new set of 256 operations different from the first "normal" set.

The extension opcode set is decoded under the assumption that all the opcodes in it are EXTs.

In this document, such extension opcodes are denoted by opcode numbers greater than 255. In effect, the EXTOP instruction (which is never seen by the game author) says to add 256 to the opcode following it. EXTOP would normally be handled during instruction decoding, rather than executing it as a real opcode.

3.3 Arithmetic Operations

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

[ADD] arg1:int,arg2:int >VAL 2OP:20 Adds the integers.

`SUB arg1:int,arg2:int >VAL` `2OP:21`

Subtracts arg2 from arg1.

`MUL arg1:int,arg2:int >VAL` `2OP:22`

Multiplies the integers.

`DIV arg1:int,arg2:int >VAL` `2OP:23`

Divides arg1 by arg2, returning the truncated quotient.

`MOD arg1:int,arg2:int >VAL` `2OP:24`

Divides arg1 by arg2, returning the remainder.

`RANDOM arg:int >VAL` `EXT:231`

Returns a random value between one and arg, inclusive. Args of zero or less are treated specially, as follows.

Random numbers may be generated by a hardware random number generator, if available, or by some other method. The most common is to update the seeds while waiting for keyboard input, as the amount of time such input will take is unpredictable. If this method is used, both READ and INPUT should employ it.

It is sometimes useful to be able to get predictable results from RANDOM. For example, playing through a game from a script in order to reach a predictable point.

Arg of a negative number makes RANDOM predictable. The absolute value of arg is saved away and RANDOM generates numbers in sequence from 1 to the absolute value of arg for the remainder of the game session.

Note that the number described above is not necessarily the number returned by RANDOM, since later calls to RANDOM will have an arg as well, and the value is always MOD arg.

RANDOM with an argument of 0 resets RANDOM to its normal state (i.e. enables randomness).

`LESS? arg1:int,arg2:int /PRED` `2OP:2`

Is arg1 less than arg2?

`GRTR? arg1:int,arg2:int /PRED` `2OP:3`

Is arg1 greater than arg2?

3.4 Logical Operations

`BTST arg1:word,arg2:word /PRED` `2OP:7`

Is every bit that is on in arg2 also on in arg1?

`BAND arg1:word,arg2:word >VAL` `2OP:9`

Bitwise logical and.

`BOR arg1:word,arg2:word >VAL` `2OP:8`

Bitwise logical or.

`BCOM arg:word >VAL` `1OP:143`
`BCOM arg:word >VAL` `1OP:248 X`

Bitwise logical complement. Note that in XZIP BCOM is a different opcode.

`SHIFT int,n >VAL` `EXT:258 X`

SHIFT performs a 16-bit logical shift on int, shifting it left n bits if n is positive, and right the absolute value of n bits if n is negative. In a logical shift, the sign bit is not propagated on rightward shifts, but rather zeroed.

`ASHIFT int,n >VAL` `EXT:259 X`

ASHIFT performs a 16-bit arithmetic shift on int, shifting it left n bits if n is positive, and right the absolute value of n bits if n is negative. In an arithmetic shift, the sign bit is propagated on rightward shifts, meaning that a negative number stays negative.

3.5 General Predicates

`EQUAL? arg1:any,arg2:any,arg3:any,arg4:any /PRED` `2OP:1,EXT:193`

Is arg1 equal to any one of arg2, arg3, or arg4? 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? arg:any /PRED` `1OP:128`

Is arg equal to zero?

3.6 Object Operations

Objects have six pieces of information associated with them that may be accessed using the following commands.

In code, references to objects will consume a single byte if the object number of the object is 255 or less, and a word otherwise.

Object number zero is a special-case pseudo-object used where an object-pointer slot is empty.

In EZIP/XZIP, each object contains 48 1-bit flags, arranged as three words, and numbered from left to right, 0 to 47 (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 numbers of other objects. Each of these slots is a word. These numbers are used to link objects together in a hierarchical structure. The LOC slot contains the number of 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 contains the number of the first object that this object contains, which is the first object in the NEXT chain.

`MOVE thing:obj,dest:obj` `2OP:14`

Put thing into dest. If thing wasn't already in dest, it should become the first object in dest. I.e., FIRST(dest) should be EQUAL? to thing after the MOVE.

`REMOVE obj` `1OP:137`

Removes obj. This means that the FIRST-NEXT chain it is a part of is relinked to no longer reference obj.

In terms of the actual code: If FIRST(LOC(obj)) equals obj, then NEXT(obj) =¿ FIRST(LOC(obj)). Otherwise, the FIRST-NEXT chain of LOC(obj) is searched to find an object (sib) for which NEXT(sib) equals obj. When that is found, NEXT(obj) =¿ NEXT(sib). Naturally, NEXT(obj) may be zero.

Finally, the LOC, FIRST, and NEXT slots of obj are zeroed.

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

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 >VAL` `1OP:131`

Return container of obj, zero if none.

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

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

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

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

`IN? child:obj,parent:obj /PRED` `2OP:6`

Is child contained in parent? More precisely, is the LOC of child equal to parent?

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

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 >VAL` `2OP:19`

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).

3.7 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. The development ZIP, using a symbol table provided by ZAP, will check table references for validity (i.e., make sure references to table offsets are within the table bounds, but non-development interpreters are not expected to do this.

Note that tables must all fall within the low 64K of the game's locations, as tables are always referenced by a direct byte number.

Offsets in tables are zero-based. The first element of a table is element zero, the second is element 1, and so on.

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

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.

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

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 by 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 >VAL` `2OP:18`

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 >VAL` `1OP:132`

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.

`INTBL? item,tbl,len:int >VAL /PRED` `EXT:247`
`INTBL? item,tbl,len:int,recspec:int >VAL /PRED` `EXT:247 X`

Tests whether item is an element of the tbl which contains len word-oriented elements. If so, it returns a pointer to that location within tbl in which item first appears (i.e. a GET of INTBL?'s returned value and zero would return item). If not, it returns zero. NOTE: This is also a predicate instruction.

In XZIP, INTBL? is given an optional fourth argument, the recspec, or record specification. This is a byte whose high bit determines whether INTBL? is comparing words (high bit 1) or bytes (high bit 0), and whose low seven bits are the record length in bytes. If not supplied or zero, defaults to 130. (202 octal, 82 hex) which is equivalent to the current usage of INTBL?. Len must not be less than zero.

Note that the len argument is now interpreted as the number of records to search, rather than the number of words. As an example, to search an input buffer for a specific character, one would invoke

```
GETB    INBUF,1 >LEN
ADD     INBUF,2 >TMP
INTBL?  CHR,TMP,LEN,1 >VAL /PRED
```

This expansion of INTBL? makes it possible to search tables of alternating keys and values, a case which is relatively common. For example, to search a lexical buffer for a specific word, we use a record length of four:

```
GETB    LEXV,1 >LEN
ADD     LEXV,2 >TMP
INTBL?  WRD,TMP,LEN,132 >VAL /PRED
```

`COPYT source:tbl,dest:tbl,length:int` `EXT:253 X`

Copies elements of source into dest until length bytes have been copied.

If dest is zero, means to zero length bytes of source.

If length is positive, copies length bytes from source to dest. In this case, the interpreter checks for overlap of source and dest. They overlap if source is less than dest and source+length is greater than dest. If overlap occurs, COPYT performs a "backwards" copy. This means that it copies from

```
source+length-1 to dest+length-1
source+length-2 to dest+length-2
etc.
```

until it has copied length bytes. Thus, a table can be copied to itself, leaving room for new elements at the beginning, non-destructively.

If length is negative, COPYT does not check for overlap, and always copies forwards. This allows some clever tricks. For example, if a number of bytes are placed in source, and source is copied to itself offset by that number of bytes, then the bytes will be duplicated in source until the length runs out. This can be used to zero a table or copy the same elements into many slots of one.

3.8 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:

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

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

Returns the value of var.

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

Sets the specified variable to any.

`ASSIGNED? opt:var /PRED` `EXT:255 X`

ASSIGNED? is true if an optional argument was supplied. It is similar to MDL ASSIGNED?, but not as general.

ASSIGNED? must work even if there has been a call out of a function. Therefore, the number of arguments (not locals!) passed to a function must be stored as part of the frame, and restored when the called function returns to the caller.

`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 /PRED` `2OP:5`

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

DLESS? var,int /PRED 2OP:4

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

3.9 I/O Operations

EZIP requires minimum terminal capabilities for I/O operations. These include upper & lowercase, 80-column width, and at least 14 lines in length.

XZIP relaxes the 80-column width restriction, allowing any width between 60 and 80 columns.

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 and the buffer is emptied 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 printed and emptied before each READ and INPUT operation (without going to the next line, if possible). When, between calls to READ or INPUT, the output in the text screen (screen 0) has filled the text area, a [MORE] prompt will be printed. A character will be read from the terminal before additional output is printed.

READ inbuf:tbl,lexv:tbl,time:int,handler:fcn EXT:228
READ inbuf:tbl,lexv:tbl,time:int,handler:fcn >VAL EXT:228 X

READ is significantly different between EZIP and XZIP.

The first phase of READ is the same in all three ZIPs. It prints and empties the output buffer, zeroes the "more" counter, and reads a line of input. Inbuf 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.

In XZIP, the second byte of inbuf is used to store the number of characters read. It also is used to tell READ where in the buffer to start putting new characters read. If a completely new buffer is to be read (the normal case, and the only case in EZIP) then the second byte of inbuf should be zeroed before READ is called.

Also, in XZIP READ returns a value, which is the terminating character that stopped the READ. This value is not stored in inbuf. This will be one of the characters in the newly defined TCHARS word. TCHARS points to a TABLE of bytes which are characters which should terminate input. The TCHARS table terminates with a zero byte. A line-feed is always a terminating character, whether it appears in the table or not. If the character 255.

appears in the TCHARS table, it means that all function keys (keys with values greater than 127.) are terminators.

If scripting is on, the input buffer is sent to the script file only if the terminating character is a carriage return. This prevents extra copies of an input line being sent when function keys are used. (Note that the protocol for this may change at some future date, probably by the addition of a "suspend scripting" command.)

Lexv is used to store results of parsing the contents of inbuf. 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 four-byte entries.

In XZIP, if the lexv argument is zero, the input accumulated by READ is not parsed. It is possible for READ to be executed, return a terminal character, the program perform some action based on that character, and then execute READ again, and so on until a full line of input is specified.

READ will fill each lexv 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 in EZIP, offset 2 in XZIP.) These last two values are used by the program in conjunction with PRINTC to reprint words. This part of READ may be invoked separately in XZIP as the LEX instruction.

READ reads text until it encounters a newline character (or any character in the TCHARS table in XZIP). If the buffer is full, the correct action would be to ring the bell when additional characters are typed. Other actions (like an assumed newline) are considered inferior implementations and should be avoided where possible. 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 9 (5-bit) bytes to fit into three machine words to match the vocabulary table entries. (Note that as in all Z strings, the high-order bit of the last (third) word will be on.) This may actually correspond to less than 9 ASCII characters. If the encoded word is less than 9 bytes, it should be padded with the pad character (5). The words in the vocabulary table are usually sorted to facilitate a binary search. This part of READ may be called separately in XZIP as the ZWSTR instruction.

The optional arguments time and handler are used to implement timed input. The optional arguments to the INPUT instruction work analogously. The first specifies the time to wait before timing out in 10ths of a second. The second specifies a routine to CALL (internally!) when the timeout occurs. If this routine returns true (1), the input operation (READ/INPUT) is aborted. If it returns false (0), the input operation continues where it

left off. Note: The intention is that the timeout routine will be short so as not to grossly interfere with the player's input.

`LEX inbuf:tbl,lexv:tbl,lexicon:tbl,preserve:bool  EXT:251 X`

Tokenizes and looks up an input buffer's contents. The first two arguments are exactly as for READ. The third argument, if not supplied, is the normal vocabulary list. If supplied, it is an additional vocabulary list.

Note that LEX is exactly like the parsing phase of READ. This means that if an additional vocabulary list is used on an input buffer that contains only words from the normal vocabulary list, it will not find them and thus will zero their slots in the lexv.

For this reason an additional argument, preserve is defined for LEX. If supplied and non-zero, it means that the lexv slots for words not found are not to be touched. Using this argument, several successive vocabulary lists can be applied to the same input buffer.

`ZWSTR inbuf:tbl,inlen:int,inbeg:int,zword:tbl  EXT:252 X`

Takes an input buffer pointer, the length of the word being converted, the character offset in the buffer of the start of the word, and a pointer to a table with at least six bytes that can be clobbered. It would also be possible to pass a RESTed inbuf and no inbeg, but this form of ZWSTR duplicates the format of a lexical buffer and is therefore preferable. ZWSTR expects the word to be terminated by one of the usual break characters, so the inlen argument is not actually needed. It is included for possible future uses. A zero byte is an acceptable break character.

The ZWSTR instruction converts the "word" contained in the buffer into a ZWORD and places the conversion in the first three words of the table.

`INPUT dev:int,time:int2,handler:fcn  EXT:246`

This returns a single input from the device specified by dev. The only defined device is the keyboard (code = 1) and the instruction returns the ASCII code for the next key pressed.

In EZIP, keys which do not have a single ASCII value are ignored, with the following exceptions (assuming that these keys exist on the target machine):

```
up-arrow        return 14
down-arrow      return 13
left-arrow      return 11
right-arrow     return  7
```

In XZIP, there is a different implementation of function keys. Function keys produce values greater than 127., that is, they have the high bit of a byte turned on. Initially, there are defined twelve function keys, four arrow keys, and two "mouse clicks".

In XZIP, function keys are accepted by both the INPUT and READ instructions.

```
up-arrow                return 129
down-arrow              return 130
left-arrow              return 131
right-arrow             return 132
keys F1-F12             return 133-144
keypad keys 0-9         return 145-154
mouse double click      return 253
mouse single click      return 254
```

On the VT100 family of machines (used with ZIP20) the keypad keys return values from 133. to 150., so that games can simulate function keys.

Note that any number of additional function keys may be defined, tailored to each machine.

The optional arguments are like those for the READ instruction and are discussed in detail there. As with the READ instruction, INPUT should clear the output buffer (if output is buffered) and zero the "more" counter.

`USL` `0OP:188`

Updates the status line now instead of waiting for the next READ. This instruction is useful only in ZIP, but has not been removed in EZIP or XZIP, although some implementations treat it as an error.

EZIP and XZIP must handle their own status lines.

`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 four. The multiplication is necessary because str in this instruction is a quad-pointer, guaranteed to point to a string that has been quad-aligned.

`PRINTB str` `1OP:135`

Like PRINT, but str here is an ordinary byte-pointer, most commonly a vocabulary table entry.

`PRINTD obj` `1OP:138`

Prints the short description of obj.

`PRINTI (in-line string)` `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 (in-line string)` `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).

`PRINTT bytes:tbl,width:int,height:int,skip:int` `EXT:254 X`

PRINTT takes a table of bytes, a width (a number of columns) and optionally a height (a number of lines), which is assumed to be one if omitted. It also optionally takes a skip, which is how many bytes of a table to skip over at the end of each line (by default, none).

It prints, in a block at the current cursor position, bytes from the table. Each group of width bytes is printed on a separate line aligned with the first, until height lines have been printed. Each time width bytes have been printed, skip bytes are skipped over. The skip parameter allows a rectangular block of text from anywhere within a rectangular table (one where the rows are stored) to be printed.

PRINTT with a height greater than one may only be used in screen 1.

`SPLIT height:int` `EXT:234`

In EZIP, if MODE bit 0 (ESPLIT) is zero, this operation is ignored. In XZIP, this operation must be implemented.

SPLIT divides the screen into two windows: #1 occupies height lines, preferably at the top of the screen, and #0 occupies the remainder of the screen. If height 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.

For example, under normal circumstances, the SPLIT opcode should cause no visible changes on the screen. However, when the cursor position in screen 0 is within the area which will become screen 1, the cursor should be moved to the top and left of the new screen 0.

`SCREEN screen:int` `EXT:235`

In EZIP, if MODE bit 0 (ESPLIT) is zero, this operation is ignored. In XZIP, this operation must be implemented.

SCREEN causes subsequent screen output to fall into window #screen.

In EZIP, if screen is 1, the output cursor is moved to the upper left-hand corner; if screen is 0, the output cursor is restored to its previous position. When moving from screen 0 to screen 1, the buffer should have previously been printed and emptied if buffering is on.

In XZIP, when either screen is departed, the position in that screen is remembered, and when the screen is reentered, it is restored. When screen 1 is first selected after a SPLIT instruction, the cursor moves to the top and left of screen 1.

Output to screen 1 is not sent to the printer.

This operation is ignored if the screen is not split, or if screen is not zero or one.

`CLEAR screen:int` `EXT:237`

In EZIP, if MODE bit 0 (ESPLIT) is zero, this operation is ignored. In XZIP, this operation must be implemented.

If screen is 1 or 0, CLEAR clears window #screen. If screen is -1, it unsplits the screen (if it has been split) and clears the entire screen. Other values for screen are ignored.

When a screen is cleared, the cursor moves to the top and left of that screen. In XZIP, the screen is cleared to the current background color.

`ERASE int` `EXT:238`

In EZIP, if MODE bit 4 (ECURS) is zero, this operation is ignored. In XZIP, this operation must be implemented.

ERASE erases the line on which the cursor lies, according to int. If int is 1, it erases from the cursor to the end of the line. There are no other legal values for int at the present time.

In XZIP, the ERASEd area should be colored the background color.

`CURSET y:int,x:int` `EXT:239`

In EZIP, if MODE bit 0 (ESPLIT) is zero, this operation is ignored. In XZIP, this operation must be implemented.

CURSET moves the cursor to line #y, column #x in screen 1.

In EZIP, this operation is illegal if the screen is not split or if screen 0 is active. CURSET is also illegal in EZIP if output is buffered (i.e. the BUFOUT instruction has not been used with a zero argument).

In XZIP, the arguments to CURSET are in pixels rather than lines and characters. The upper left corner of the screen is the origin, and is referred to as 1,1.

If there are no variable width fonts or pictures on a particular micro, the XZIP may be implemented as though characters were one pixel wide and one pixel high.

In XZIP, CURSET must output any buffered output before actually moving the cursor. The requirement that buffering be turned off before CURSET is executed is removed.

`CURGET` `EXT:240`
`CURGET output:tbl` `EXT:240 X`

In EZIP, this is not currently implemented, although the operation is reserved.

In XZIP, returns information about the current cursor position. It is passed an output table which must have the first two words free to write in. CURGET writes the y position in the word 0 of the table, and the x position in word 1 of the table. The positions are as for CURSET.

`HLIGHT int` `EXT:241`

HLIGHT sets the display highlighting mode for all subsequent output. Some machines may not be able to do all highlighting modes, and MODE bits 1 (EHINV), 2 (EHBLD), and 3 (EHUND) determine which are available. If the appropriate option bit in the mode byte is zero, HLIGHT is ignored. Otherwise, it is interpreted as follows:

```
0 - no highlight
1 - inverse video
2 - bold
4 - underline or italic at the interpreter's discretion.
8 - monospaced font (XZIP only)
```

Note that the codes are set up as powers-of-two. This is intentional, but it is NOT required at this time that the interpreter handle combination highlights (bold + italic).

A note regarding the "monospace" highlight. It either selects a monospaced font if one is available, or modifies the screen display of a variable width font so that it appears

monospaced.

If the intent of using monospacing is to do something like tabs (i.e., go to some point on the screen and then print stuff), then CURSET and a variable width font are better. Use of the monospace highlight mode should be reserved for cases (like the Translucent Maze in Enchanter) where all of the columns must line up.

`FONT font:int >VAL` `EXT:260 X`

Selects a particular font for the currently selected window, and returns the number of the previously selected font. If the new font cannot be selected for some reason, returns 0. The font should be remembered for that window until it is explicitly changed. Font 1 is the "normal" font for the machine in question, and it is selected initially for both screen windows. The interpreter is responsible for updating the FWRD parameter word whenever the font changes.

FONT prints and empties the output buffer.

It should be possible to change fonts many times, even during a line or word of output.

In ZIP20, in addition to the normal font (1), and the picture font (2), font 3 is the VT100 character graphics set.

`COLOR fore:int,back:int` `2OP:27 X`

In XZIP, if MODE bit 0 (XCOLOR) is zero, this operation is ignored. In EZIP, this operation does not exist.

COLOR causes the foreground color of all subsequently displayed text to be fore, and the background color to be back.

COLOR prints and empties the output buffer.

The screen data word CLRWRD contains the system default colors. The background color in the first byte and the foreground color in the second byte.

The values of fore and back are interpreted as follows (the colors are those for the IBM-PC):

```
0 = no change
1 = system default color
2 = black
3 = red
4 = green
5 = yellow
6 = blue
7 = magenta
8 = cyan
9 = white
```

Note that the colors available may vary from machine to machine. We have to decide on either a set of colors common to all machines, or admit that the set will vary in each interpreter. Also, not all machines can handle this many colors.

On the Atari ST, XZIP uses:

```
- medium res
- any 4 colors, selected at runtime
- 640 (80 cols) x 200
```

The CLEAR opcode is defined to clear the window(s) to the background color specified by the last COLOR opcode, and the ERASE opcode similarly to erase to the background color.

Games should allow for primitive machines (Apple IIs, Macs and the 20) that don't have color. Machines with monochrome display options should ask the player whether he or she wants color.

3.10 Static Graphics

The current EZIP specification requires host machines to emulate the 80-column alphanumeric display of a VT100. It does not exploit or even acknowledge the bitmap displays employed by most EZIP micros. The following additions allow XZIP game designers to take advantage of some of this lost potential.

3.10.1 Pictures

Pictures are not necessarily included in the virtual address space of the game, and may be stored in machine-dependent ways. It is the job of the interpreter to map between picture references in the game and picture storage of whatever sort.

```
DISPLAY      picture:int,x:int,y:int                          EXT:261 X
```

A picture is a number that indexes into the "picture library."

DISPLAY displays a picture at the location (x,y) (specified in pixels). The location given is where the upper left corner of the picture should appear. The upper left corner of the screen is the location 1,1.

If the x or y argument is not supplied or 0, then the current x or y position in the current window is used.

If (and only if) both x and y are omitted, DISPLAY updates the cursor position in the window in which the picture is displayed. In this case, the new cursor position will be the upper right of the picture displayed. In other words, the x position is updated by the width

of the character, and the y position is unchanged. Consequently, passing a single argument is very similar to displaying the picture as though it was a character.

If DISPLAY is used in this mode, the picture displayed should be treated like a character (the one with the same ascii value as the picture's picture number) for purposes of scripting the output. However, pictures DISPLAYed with explicit location arguments are not scripted.

It is expected that in normal use, pictures that are "like characters" will be displayed primarily in window 0, and pictures that are "like pictures" will be displayed primarily in window 1.

`PICINF picture:int,data:tbl /PRED` `EXT:262 X`

PICINF is used to get data about a picture. The interpreter fills in the table data with the width (word 0) and height (word 1) of the picture specified, in pixels. It is up to the interpreter to determine from the picture image itself the width and height of the picture.

Since zero is not a legal picture id, if the picture argument to PICINF is zero, then the highest picture id in the picture library will be returned in word 0 of the data table. This will be equivalent to the number of pictures in the library if all ids are used, but there is no requirement that this be the case.

If the picture number given is not a legitimate picture number, PICINF returns false. Note that DISPLAY and DCLEAR give errors in this situation!

See the section 4.7 for an example specification of how pictures are stored.

`DCLEAR picture:int,x:int,y:int` `EXT:263 X`

Clears the area taken up by the picture. I.e., restores the screen background color.

There should probably be some pictures defined that would be used only for DCLEAR, such as a picture of the whole screen.

*Note: If possible, we should define DISPLAY to save the previous contents of the area overwritten by the new picture. If so, then a DCLEAR should restore the overwritten area. There may have to be a limit on how large a picture could be saved this way, and how long to remember it. (It may be impossible to do this on some machines.)

3.10.2 Interaction of Pictures and Fonts

The "picture library" is a selection of images which are referenceable using small integers as an index. In this way, a picture library is very much like a font. In a font, the images are characters, and in a picture library they are pictures. Consequently, font 2 is defined to be the picture library. If FONT 2 is executed, subsequent character output to the screen is displayed from the picture library. If there is no picture library, then FONT 2 is a no-op.

*Note: Do pictures need a way of saying that the horizontal and vertical position shouldn't change when they are displayed?

3.10.3 Changing Margins

`MARGIN left:int,right:int` `EXT:264 X`

Sets left margin and right margin in pixels. Left and right are the width of the margins, not the locations of the margins, so both are initially 0. The margins are stored by MARGIN in the LMRG and RMRG words.

MARGIN can only be executed in screen 0 (since only screen 0 has text folding). It must be executed before any text has been buffered for the current line, and moves the cursor to the new left margin on the current line.

If there is a non-zero left margin, clearing screen zero should position the cursor at the left margin of the top line of screen 0.

The margin is used to control insets (as for the proposed "Illuminated Text Adventure").

3.10.4 Carriage Return Interrupt

Two new words, CRCNT and CRFUNC, are defined. Before the interpreter outputs a carriage return, it checks CRCNT, and if it is non-zero, decrements it. If CRCNT reaches zero after such an operation, the contents of CRFUNC are called as a function address. This feature can be used to set up indenting around pictures.

3.11 Misc. I/O Operations

`BUFOUT int` `EXT:242`

Determines whether or not output is line-buffered. If int is 1 (the normal case), output is buffered a line at a time so that line breaks can be planned for. If int is 0, all currently buffered output is sent to the screen, and all future output is sent to the screen as it is generated.

The "line position" counter should NOT be cleared when a BUFOUT of 0 is performed. In this way, when buffered output is re-enabled, line position is not lost.

In EZIP, disabling buffered output MUST be performed prior to using the CURSET opcode.

Note: Output redirected to a TABLE (see next instruction) is not buffered.

In XZIP, the interactions between buffering and certain opcodes are slightly different, so BUFOUT is not as necessary. The CURSET, SCREEN, ERASE, CLEAR and COLOR opcodes output any buffered text to the screen before performing any other action. Any future opcodes defined which may change screen appearance will also be defined to output buffered text as their first action.

Depending on how SPLIT, CURGET, DIROUT, FONT and HLIGHT are implemented in a particular interpreter, they may also want to output buffered text. In some implementations, they will not need to, as for example HLIGHT in ZIP20 is implemented by placing highlighting characters in the output stream. In an interpreter where HLIGHT is done by a system call (one meaning "future output is underlined,") it should output the buffer first.

`DIROUT device:int,any1,any2,any3` `EXT:243`

Selects or deselects a virtual output device according to device. Each virtual device is assigned a code, and the game indicates its desire to select or deselect that device by passing a first argument of device or minus device, respectively.

Device 1 is the screen and is the default output device. It can be shut off by passing -1 to DIROUT.

Device 2 is the transcript device. (This interface replaces the ZIP method of setting a bit in the mode word.) It sends a transcript of player input and all output in screen 0 to a transcript, which may be a file, a printer, or any appropriate device. Transcripting is terminated when device 2 is deselected. When the interpreter is transcripting, it should set bit 0 (FSCRI) in the FLAGS word.

Note that if the screen device is off and the transcript device is on, output goes to the transcript device anyway. In this way, text can be placed in the transcript without it having to appear on the screen. This is useful for copying screen 1 output to the transcript.

Note that in XZIP, the existence of a resumable READ instruction implies that the input buffer read by READ may need to be output manually, or the script file will end up with a copy of the input buffer each time READ returns.

Device 3 is the table output device. It directs output to the TABLE specified as any1. Each character to be printed is PUTB'd into the TABLE starting at TABLE+2. When a carriage-return line-feed is printed, a 13 (hex $0D) is placed in the table. When deselected, the number of characters placed into the TABLE will be PUT into the 0'th element of TABLE. Output redirected to a TABLE is not buffered. When the table device is selected, all other devices are ignored until it is deselected.

Device 4 is the command recording device. It creates a command file which consists of the commands input to the game via READ and INPUT. The file is closed when device 4 is deselected. Note that this device is currently optional. An interpreter which does not handle this device should ignore the request for selection and deselection.

DIRIN device:int,any1,any2,any3 EXT:244

Redirects input according to device.

Device 0 is the keyboard (this is the default case).

Device 1 is a command file (such as use of DIROUT 4 might produce). Input is received from the command file (this need not be implemented on all interpreters, but might be useful for running scripts).

No other values of int are legal at this time.

SOUND int EXT:245
SOUND id:int,op:int EXT:245 (X)

If the appropriate bit in the mode byte is zero, this operation is ignored. Otherwise, produce the sound specified by int.

In EZIP, the following sounds are defined:

```
1 - beep (equivalent of a morse code dot)
2 - boop (equivalent of a morse code dash)
```

In XZIP (and the Amiga version of ZIP), a new sound specification exists. SOUND takes the same sound-identifier argument as before, but adds a sound-operation argument as well. Currently, there are only three operations defined:

op	meaning
1	initialize specified sound
2	start specified sound
3	stop specified sound
4	clean up buffers from specified sound

If the sound-id is 1 or 2 (beep and boop), the op is ignored. If the sound-id is 0, the last sound-id specified is used. If no op is supplied, op 2 (start) is assumed.

3.12 Control Operations

CALL fcn,any1,any2,any3 >VAL EXT:224

Begins execution of the function (see Chapter 4) pointed to by fcn times four, supplying it with any arguments given in the CALL instruction. Note that fcn is a quad-pointer and functions are always quad-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.

`CALL1 fcn >VAL` `1OP:136`

Same as CALL, but a 1-op.

`CALL2 fcn,any >VAL` `2OP:25`

Same as CALL, but a 2-op.

`XCALL fcn,any1,any2,any3,any4,any5,any6,any7 >VAL` `EXT:236`

Same as CALL, but with 4-7 arguments supplied. This instruction is never invoked with fewer than 4 arguments.

`ICALL1 routine:fcn` `1OP:143 X`

`ICALL2 routine:fcn,arg1:any` `2OP:26 X`

`ICALL routine:fcn,arg1:any,arg2:any,arg3:any` `EXT:249 X`

`IXCALL routine:fcn,arg1,...` `EXT:250 X`

In XZIP, there are versions of the CALL instructions which do not return a value. ICALL, ICALL1, ICALL2, and IXCALL are defined exactly as their counterparts CALL, etc., except that they do not return anything. The return byte is therefore omitted. These opcode are generated by the compiler when it notices that the value of a routine is unused. This has the advantage of reducing stack usage and limiting stack overflows.

Note that the interpreter must remember that a valueless call was executed, and this information must be immediately saved as part of the routine's state information.

`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."

CATCH >VAL 0OP:185 X
THROW any,frame 2OP:28 X

CATCH returns a pointer (called a frame) to the call to the current routine. THROW returns any from a frame. It is as though the routine in which the CATCH was done returned any. The frame should be one that is still "alive," meaning that when the THROW is executed, it is in a routine called (directly or indirectly) by the routine that did the CATCH.

CATCH and THROW are not defined to work within "internal" calls, such as the timeout handling routine that can be called by READ or INPUT.

Note that the opcode for CATCH recycles the former opcode for FSTACK.

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. This instruction does not exist in XZIP, as no-value CALLs eliminate the need for it.

NOOP 0OP:180

No operation, equivalent to a "JUMP 2."

3.13 Game Commands

SAVE >VAL OOP:181
SAVE start:tbl,length:int,name:tbl >VAL EXT:256 X

In EZIP, 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.

In XZIP, SAVE (and RESTORE) may be used as atomic i/o operations.

SAVE writes a section of the impure area. The length argument is the length of the save area in bytes.

The name argument is a one-byte count of bytes in the name, followed by that number of bytes of name. It is the game's unique name for the file being created. RESTORE should check that the name is the same in the file being restored as the RESTORE's name argument. If it is not, it is an error.

It is expected that the player will be called upon to supply a file name or number or whatever. This may well be the same as the name argument on machines with file systems, but need not be. It is recommended that the name argument be displayed or used as a default when the player is consulted, however.

SAVE returns a value as before. It returns zero if it failed, 1 after SAVE, and 2 after RESTORE (as RESTORE merely causes a SAVE to "return again").

In high-level terms, the game saves or restores a TABLE or a group of contiguous tables. Some additions may be needed to ZIL/ZILCH to force a set of tables to be compiled contiguously.

RESTORE >VAL OOP:182
RESTORE start:tbl,length:int,name:tbl >VAL EXT:257 X

In EZIP, 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.

In XZIP, RESTORE may be used as an atomic i/o operation. RESTORE reads a section of the impure area.

See SAVE for details about the other arguments.

RESTORE returns the number of bytes read if called with three arguments ("partial RESTORE"), returns zero if it fails, and otherwise doesn't return (the SAVE "returns again").

`ISAVE >VAL` `EXT:265`

This instruction copies the impure area to a reserved part of RAM where it can be copied back by the IRESTORE command. It returns 0 if it fails or -1 if the instruction is not implemented on the machine.

ISAVE and IRESTORE, in combination, allow an UNDO command to be implemented.

`IRESTORE >VAL` `EXT:266`

This instruction causes the saved copy of the impure area to be copied back to the impure area, and thus is a single level UNDO command. It returns 0 if it fails or if the instruction is not implemented on the machine. If an ISAVE has never been executed successfully, IRESTORE should return 0.

`VERIFY /PRED` `OOP:189`

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*4-1, with PCHKSM. Note that for the preloaded area, the unmodified pages on the disk should be used rather than the pages in RAM

`ORIGINAL? /PRED` `OOP:191`

Returns non-false if the game disk is the original. Implementation is unspecified.

`RESTART` `OOP:183`

Reinitializes the game, reloads the preload area from disk, 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 (those marked with a * are writeable by game code, the others may only be read):

word	name	used for
0	ZVERSION	version of Z-machine used
1	ZORKID	version of game
2	ENDLOD	beginning of non-preloaded code
3	START	location where execution begins
4	VOCAB	points to vocabulary table
5	OBJECT	points to object table
6	GLOBALS	points to global variable table
7	PURBOT	beginning of pure code
8	*FLAGS	16 game-settable flags
9	SERIAL	serial number - 6 bytes
12	FWORDS	points to fwords table
13	PLENTH	length of program (in quads)
14	PCHKSM	checksum of all bytes
15	INTWRD	interpreter identification word
16	SCRWRD	screen parameters word
(the following words are XZIP-only)		
17	HWRD	width of display in pixels
18	VWRD	height of display in pixels
19	FWRD	one byte font height, one font width
20	LMRG	left margin in pixels
21	RMRG	right margin in pixels

```
22      CLRWRD          one byte background color, one foreground color
23     *TCHARS          pointer to table of terminating characters
24     *CRCNT           counter for carriage returns
25     *CRFUNC          function for carriage returns
26      CHRSET          pointer to character set table
27      EXTAB           points to extension table, if needed
28-31   USRNM            eight bytes of user name (ZIP20 only)

(extension table words)

0                       length of extension table
1       MSLOCX          x location of mouse
2       MSLOCY          y location of mouse

(the following words are defined but no XZIP implements them)

3      *MSETBL          mouse table change word
4      *MSEDIR          direction menu
5      *MSEINV          inventory menu
6      *MSEVRB          frequent verb menu
7      *MSEWRD          frequent word menu
8      *BUTTON          button handler
9      *JOYSTICK        joystick handler
10     *BSTAT           button status
11     *JSTAT           joystick status
```

ZVERSION is interpreted as two bytes, VERSION and MODE. All games produced in XZIP will have a Z-machine version byte of 5; EZIP games will have a version byte of 4; ZIP games will have a version byte of 3. Combined XZIP/EZIP/ZIP interpreters will need to have this information, of course. The mode byte contains eight option bits which vary considerably between EZIP and XZIP.

```
        bit #   name    EZIP interpretation

        0       %ESPLIT SPLIT/SCREEN/CLEAR available (0 = no)
        1       %EHINV  Inverse video available (0 = no)
        2       %EHBLD  Bold available (0 = no)
        3       %EHUND  Italic/underline available (0 = no)
        4       %ECURS  CURSET/CURGET available (0 = no)
        5       %ESOUN  SOUND available (0 = no)
        6               reserved
        7               reserved

        bit #   name    XZIP interpretation

        0       %XCOLOR COLOR operation available (0 = no)
```

```
1       %XDISPL DISPLAY operation available (0 = no)
2       %XBOLD  Bold available (0 = no)
3       %XUNDE  Italic/underline available (0 = no)
4       %XMONO  Monospace style available (0 = no)
5       %XSOUN  SOUND available (0 = no)
6               reserved
7               reserved
```

Note that this byte is set by either a loader for a particular machine or the interpreter at start-up time.

ZORKID is the version of the game. This is what is usually printed by a game as the "release number."

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 RAM. 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 256k 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.

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

```
bit #   name    interpretation

0       %FSCRI  interpreter currently transcripting
1       %FFIXE  fixed-width font needed (EZIP)
2       %FSTAT  request for status line refresh
3       %FDISP  game uses display operations
4       %FUNDO  game uses UNDO
5       %FMOUS  game uses mouse
6       %FCOLO  game uses colors
7-15            reserved
```

Bit #1 (%FFIXE, EZIP only) 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.

Bit #2 (%FSTAT) should be set by the interpreter whenever, in its opinion, the status line area has become damaged or is suspect (perhaps due to target machine operating system intervention). The game is responsible for refreshing the status line area (if any) and will also clear this bit when the refresh is completed.

Bit #3 (%FDISP) should be set at compile time by games which will be using the display

and graphics operations, such as DISPLAY, DCLEAR, and FONT. This is because some interpreters will choose to be in a "graphics" mode if these operations are used, and a "text" mode otherwise. If the game chooses to be in graphics mode, and the interpreter cannot support graphics mode, it clears this bit at initialization time.

Bit #4 (%FUNDO) should be set at compile time by games which will try to use ISAVE and IRESTORE. The interpreter should examine this bit in marginal memory size cases to determine how many swapping pages to allocate.

Bit #5 (%FMOUS) should be set by games which use the mouse.

Bit #6 (%FCOLO) should be set by games which will be using color (through the COLOR operation). Some interpreters (currently only the Amiga) will examine %FCOLO at startup, since they have to allocate extra memory for the display RAM if color is needed. In any case, %XCOLOR in the MODE byte will be set or cleared depending on whether the interpreter supports the COLOR operation.

SERIAL is a six-character ASCII string intended to uniquely identify each copy of a game. This string was to be inserted when each distribution disk is created and read by the game program when executed. In practice, it contains the date the release of the game was compiled.

PLENTH and PCHKSM are both used by the VERIFY operation. PCHKSM is the 16-bit sum of all bytes from 64 (decimal) to PLENTH*4-1.

INTWRD is composed of 2 bytes, called INTID and INTVR. The high byte is the interpreter id, an integer unique for a given interpreter. Currently assigned intids include:

machine	id #
DECSystem-20	1
Apple IIe	2
Macintosh	3
Amiga	4
Atari ST	5
IBM PC	6
Commodore 128	7
Commodore 64	8
Apple IIc	9
Apple IIgs	10

The low byte is the interpreter version identifier, an ASCII character which identifies the release of the given interpreter. By convention, these are letters of the alphabet starting with A. This word is set by the interpreter upon initialization.

SCRWRD is composed of 2 bytes, called SCRV and SCRH. SCRV indicates the number of lines available on the screen (255 meaning a printing terminal), and SCRH indicates the number of characters on a line. This word is set by the interpreter upon initialization.

HWRD indicates the width of the screen in pixels.

VWRD indicates the height of the screen in pixels.

FWRD consists of two bytes, called FNTV and FNTH. FNTV is the vertical size of the current font, in pixels. FNTH is the horizontal size of the the current font (in a variable width font this would be the size of a digit character).

LMRG contains the left margin (initially 0) in pixels, as set by MARGIN.

RMRG contains the right margin (initially 0) in pixels, as set by MARGIN.

CLRWRD consists of a high byte which gives the default screen background color, and a low byte which give the default screen foreground color.

TCHARS is a pointer to a byte table (whose address must be below ENDLOD) terminated by a zero byte. It is used by READ to decide which input characters to terminate on. The bytes contain the characters to terminate on. An end-of-line always terminates, whether it is explicitly in the table or not. A 255 in the table indicates that all function keys terminate.

CRCNT, if non-zero, is decremented whenever a CRLF is output.

CRFUNC is called when CRCNT reaches zero.

CHRSET is used to define the printing character set. It is not currently implemented in any XZIP. This word points to a table of 78 bytes. The first 26 characters in it form character set 0, the second character set 1, and the third character set 2. All other characters would be represented using the ascii escape sequence. Space is defined to be in all character sets.

EXTAB, if non-zero, is a pointer to an LTABLE containing any of the remaining words that are used, if any.

MSLOCX, when a mouse operation occurs, will contain the X coordinate of the mouse, in pixels. Mouse single and double clicks are the only mouse operations supported.

MSLOCY, when a mouse operation occurs, will contain the Y coordinate of the mouse, in pixels.

MSETBL contains bits which are set to indicate that the game has changed one of the mouse menu tables.

MSEDIR contains a pointer to the mouse direction menu.

MSEINV contains a pointer to the mouse inventory menu.

MSEVRB contains a pointer to the mouse frequent verbs menu.

MSEWRD contains a pointer to the mouse frequent words menu.

BUTTON contains a pointer to the function which will handle button events.

JOYSTICK contains a pointer to the function which will handle joystick events.

BSTAT contains the state of the buttons.

JSTAT contains the state of the joystick.

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 ZIP 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 63 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 63) 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. An object is formatted as follows:

byte	value
0-1	first flag word, flags 0-15
2-3	second flag word, flags 16-31
4-5	third flag word, flags 32-47
6-7	LOC slot
8-9	NEXT slot
10-11	FIRST slot
12-13	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 or 2 bytes)
property value (1-64 bytes)
.
.
.
property identifier
```

```
property value
0
```

There may be from 0 to 63 property pairs. Each property identifier has the property number in the low-order 6 bits. The high-order bit, if set, indicates that there are more than 2 bytes in the property value, in which case the following byte will have the two high bits set and the low-order 6 bits will be the length of the property value. Otherwise, the second-high bit (64 bit) will be on for a length of 2 bytes, off for a length of 1 byte. For searching efficiency, the properties are sorted in inverse order by property number. [Note: The two high bits are set in the extended property length byte so that PTSIZE can be implemented properly. Otherwise, it would be impossible to interpret the byte preceding the start of the property value.]

4.4 Vocabulary Table

This table contains the words that will be understood by READ (or LEX), other information for READ, and, optionally, some game-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 (6-byte string)
extra entry bytes for word #1
.
.
.
word #m
extra entry bytes for word #m
```

In XZIP, only the main vocabulary table (the one pointed to by VOCAB) will be looked at to find the self-inserting break characters. Other vocabularies should contain 0 as the count of self-inserting breaks.

The format for number of entries in the lexicon determines whether the lexicon is sorted. If the number of entries is positive, then the lexicon is sorted. Otherwise, the lexicon is unsorted, and contains the absolute value of the number given entries.

Words are truncated or padded to cause them to fit into 6 bytes. READ performs the same function, so comparisons work. Words in the vocabulary table are sorted according to this 6-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. Only the last word in each string has the high-order bit set. If the last word is not filled, it is padded with the standard pad character (5), which conveniently doesn't print anything.

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

Assuming that the CHRSET word is zero (as it always is in current usage), the character sets are as follows:

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 The Frequent Words Table

The frequent words table, pointed to by FWORDS and below ENDLOD (a pure table), contains 96 quad-pointers to ordinary strings. These strings represent frequently used sub-strings (usually words) within other strings. Whenever a 1, 2, or 3 byte is encountered in a string that 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 (7 with the XCALL instruction).

A function may be preloaded or disk-resident (or both). It begins on a quad-boundary. The first byte specifies the number of local variables to be used by the function (0 to 15).

In EZIP, this byte is followed by one word for each such variable giving its initial or default value. Up to seven of the local variables may be initialized to values supplied in the CALL instruction instead of these. Note that this format allows for optional arguments, but only if the default value of the optional argument is a constant.

In XZIP all locals are initialized to zero by default, and there are no value words. Any local not initialized to zero (in the higher-level ZIL code) will be set up in the body of the function. Any optional arguments with non-zero default values are expected to use ASSIGNED? (q.v.) to determine whether to use the default values.

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.

Information that must be preserved over functions calls include the values of local variables in calling functions, the state of the stack when the call was performed, the number of arguments passed to the calling routine, and whether the calling routine is expected to return a value.

4.7 Graphic Data Files

This is one specification for the format of an XZIP graphic data file. It is not intended to be mandatory, but rather an example of how to store graphic data.

Graphic data can be either pictures, or an alternate character set. An alternate character set is a special case of a picture set: all entries are the same size. Some micro OSes may

provide support for alternate characters without switching into graphics mode.

4.7.1 Picture Library Header

The file header format consists of 16 words of general information.

word	name	interpretation
0	GCOUNT	number of picture/character defs in file
1	GOFF	offset of the initial p./c.
2	GSIZE	zero, or size of alternate characters
3	GFLAGS	special picture library information
4-15		reserved

GOFF allows alternate character reference numbers to start higher than zero (at 32, for example).

GSIZE, if zero, indicates that the file is a picture library, rather than a character set. If non-zero, the first byte is the width in bits of each character, the second byte is the height in bits.

GFLAGS permits 16 special purpose flags. Currently, only bit 0 is defined. If set, the interpreter is to display the first entry in the picture library while the rest of the game is loading.

4.7.2 Picture Library Data

If the file is of alternate characters (in other words, if the file is a font definition), the rest of the file contains the character definitions.

If the file is a picture file, there are two or three tables, each containing picture information.

Size table. Contains GCOUNT two-word entries. The first (XPIC) word of each entry is the width in bits of the corresponding picture. The second (YPIC) word is the height in bits of the same picture. Two zeroes is a legal entry, and means that the corresponding picture is not present.

Name table. Contains GCOUNT twelve-byte entries. Currently, the DOS filenames of the corresponding pictures.

Index table. Currently absent, would contain GCOUNT four-byte entries. Byte offset, from the beginning of the file, of the corresponding picture data.

Picture data. The remainder of the file is picture data. If XPIC for a picture is not a multiple of sixteen, the LSBs of the last word in each row will be skipped over. In other words, the bits in a row always begin at the beginning of a word. @newpage

4.8 Differences Between ZIP and EZIP

The major differences between ZIP and EZIP have to do with supporting a large address space for games, and with expanding the numbers of objects, properties, and flags in a game.

Some of the more important features of ZIP:

- A maximum program size of 128K bytes.
- A maximum of 255 objects. All object references are one byte. The length of an object is nine bytes.
- A maximum of 32 flag bits.
- A maximum of 32 properties.
- Vocabulary words contain only the first 6 characters of the word.
- Functions may be passed a maximum of 3 arguments.
- Functions and global strings must fall on word boundaries, and are referenced by a word address.
- The status line's appearance is embedded in the interpreter, and only two types of status line exist; a "score" status line and a "time" status line.
- There is no screen control, except for two opcodes, SPLIT and SCREEN.

4.9 ZIP Opcode Summary

What follows is an alphabetically arranged list of all ZIP opcodes. Note that some opcode names appear twice, as they differ significantly between EZIP and XZIP. In such cases, the XZIP version of the opcode is followed by an X.

```
ADD arg1:int,arg2:int >VAL                                  2OP:20
ASHIFT  int,n >VAL                                          EXT:259 X
ASSIGNED? opt:var /PRED                                     EXT:255 X
BAND arg1:word,arg2:word >VAL                               2OP:9
BCOM arg:word >VAL                                          1OP:143
BCOM arg:word >VAL                                          1OP:248 X
BOR arg1:word,arg2:word >VAL                                2OP:8
BTST arg1:word,arg2:word /PRED                              2OP:7
BUFOUT int                                                  EXT:242
CALL fcn,any1,any2,any3 >VAL                                    EXT:224
CALL1 fcn >VAL                                              1OP:136
CALL2 fcn,any >VAL                                          2OP:25
CATCH >VAL                                                  0OP:185 X
CLEAR screen:int                                            EXT:237
COLOR fore:int,back:int                                     2OP:27 X
COPYT source:tbl,dest:tbl,length:int                        EXT:253 X
CRLF                                                        0OP:187
CURGET                                                      EXT:240
CURGET output:tbl                                           EXT:240 X
CURSET y:int,x:int                                          EXT:239
DEC var                                                     1OP:134
DIRIN device:int,any1,any2,any3                                 EXT:244
DIROUT device:int,any1,any2,any3                        EXT:243
DIV arg1:int,arg2:int >VAL                                  2OP:23
DLESS? var,int /PRED                                        2OP:4
EQUAL? arg1:any,arg2:any,arg3:any,arg4:any /PRED            2OP:1,EXT:193
ERASE int                                                   EXT:238
EXTOP opcode:int                                            0OP:190 X
FCLEAR obj,flag                                             2OP:12
FIRST? obj >VAL /PRED                                       1OP:130
FONT font:int >VAL                                          EXT:260 X
FSET obj,flag                                               2OP:11
FSET? obj,flag /PRED                                        2OP:10
FSTACK                                                      0OP:185
GET table,item >VAL                                         2OP:15
GETB table,item >VAL                                        2OP:16
GETP obj,prop >VAL                                          2OP:17
GETPT obj,prop >VAL                                         2OP:18
GRTR? arg1:int,arg2:int /PRED                               2OP:3
```

```
HLIGHT int                                                  EXT:241
ICALL routine:fcn,arg1:any,arg2:any,arg3:any                EXT:249 X
ICALL1 routine:fcn                                          1OP:143 X
ICALL2 routine:fcn,arg1:any                                 2OP:26 X
IGRTR? var,int /PRED                                        2OP:5
IN? child:obj,parent:obj /PRED                              2OP:6
INC var                                                     1OP:133
INPUT dev:int,int2,fcn                                      EXT:246
INTBL? item,tbl,len:int >VAL /PRED                          EXT:247
INTBL? item,tbl,len:int,recspec:int >VAL /PRED                  EXT:247 X
IRESTORE >VAL                                               EXT:266
ISAVE >VAL                                                  EXT:265
IXCALL routine:fcn,arg1,...                                 EXT:250 X
JUMP    loc                                                 1OP:140
LESS? arg1:int,arg2:int /PRED                               2OP:2
LEX inbuf:tbl,lexv:tbl,lexicon:tbl,preserve:bool            EXT:251 X
LOC obj >VAL                                                1OP:131
MOD arg1:int,arg2:int >VAL                                  2OP:24
MOVE thing:obj,dest:obj                                     2OP:14
MUL arg1:int,arg2:int >VAL                                  2OP:22
NEXT? obj >VAL /PRED                                        1OP:129
NEXTP obj,prop >VAL                                         2OP:19
ORIGINAL? /PRED                                             0OP:191
POP var                                                     EXT:233
PRINT str                                                   1OP:141
PRINTB str                                                  1OP:135
PRINTC int                                                  EXT:229
PRINTD obj                                                  1OP:138
PRINTN int                                                  EXT:230
PRINTR (in-line string)                                     0OP:179
PRINTT bytes:tbl,width:int,height:int                           EXT:254 X
PTSIZE table >VAL                                           1OP:132
PUSH any                                                    EXT:232
PUT table,item,any                                          EXT:225
PUTB table,item,any                                         EXT:226
PUTP obj,prop,any                                           EXT:227
QUIT                                                        0OP:186
RANDOM arg:int >VAL                                         EXT:231
READ inbuf:tbl,lexv:tbl,time:int,handler:fcn >VAL       EXT:228 X
READ inbuf:tbl,lexv:tbl,time:int,handler:fcn                EXT:228
REMOVE arg:obj                                              1OP:137
RESTART                                                     0OP:183
RESTORE >VAL                                                0OP:182
RESTORE start:int,length:int,name:tbl >VAL              EXT:257 X
RETURN any                                                  1OP:139
```

```
RFALSE                                                    0OP:177
RSTACK                                                    0OP:184
RTRUE                                                     0OP:176
SAVE >VAL                                                 0OP:181
SAVE start:int,length:int,name:tbl >VAL                       EXT:256 X
SCREEN screen:int                                         EXT:235
SET var,any                                               2OP:13
SHIFT   int,n >VAL                                        EXT:258 X
SOUND int                                                 EXT:245
SPLIT height:int                                          EXT:234
THROW any,frame                                           2OP:28 X
USL                                                       0OP:188
VALUE var >VAL                                            1OP:142
VERIFY /PRED                                              0OP:189
XCALL fcn,any1,any2,any3,any4,any5,any6,any7 >VAL   EXT:236
ZERO? arg:any /PRED                                       1OP:128
ZWSTR inbuf:tbl,inlen:int,inbeg:int,zword:tbl             EXT:252 X
```

4.10 Opcode Differences, EZIP vs. XZIP

The following XZIP opcodes are changed from their EZIP definitions or newly defined in XZIP.

changed opcodes

```
READ    inbuf:tbl,lexv:tbl,time:int,handler:fcn >VAL        EXT:228
HLIGHT  monospace:8                                        EXT:241
CURSET  y:int,x:int                                        EXT:239
CURGET  output:tbl                                         EXT:240
SCREEN  screen:int                                         EXT:235
ERASE   op:int                                             EXT:238
CLEAR   screen:int                                         EXT:237
SAVE    start:int,length:int,name:tbl >VAL          EXT:256
RESTORE start:int,length:int,name:tbl >VAL          EXT:257
INTBL?  item,tbl,len:int,recspec:int >VAL /PRED     EXT:247
```

new opcodes

```
EXTOP   opcode:int                                         0OP:190
LEX     inbuf:tbl,lexv:tbl,lexicon:tbl,preserve:bool       EXT:251
ZWSTR   inbuf:tbl,inlen:int,inbeg:int,zword:tbl            EXT:252
ICALL1  routine:fcn                                        1OP:143
ICALL2  routine:fcn,arg1:any                               2OP:26
ICALL   routine:fcn,arg1:any,arg2:any,arg3:any             EXT:249
IXCALL  routine:fcn,arg1,...                               EXT:250
ORIGINAL? /PRED                                            0OP:191
COPYT   source:tbl,dest:tbl,length:int                     EXT:253
PRINTT  bytes:tbl,width:int,height:int                         EXT:254
COLOR   fore:int,back:int                                  2OP:27
CATCH   >VAL                                               0OP:185
THROW   any,frame                                          2OP:28
ASSIGNED? opt:var /PRED                                    EXT:255
SHIFT   int,n >VAL                                         EXT:258
ASHIFT  int,n >VAL                                         EXT:259
FONT    font:int >VAL                                      EXT:260
DISPLAY picture:int,x:int,y:int                            EXT:261
PICINF  picture:int,data:tbl /PRED                         EXT:262
DCLEAR  picture:int,x:int,y:int                            EXT:263
MARGIN  left:int,right:int                                 EXT:264
ISAVE >VAL                                                 EXT:265
IRESTORE >VAL                                              EXT:266
```

opcode changed and recycled

BCOM int >VAL EXT:248

opcode removed and recycled

FSTACK OOP:185