ZIP

ZIP: Z-language Interpreter Program

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

3/23/89

"Two thumbs up!"

— Syscon & Ebozz

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. YZIP, the most recent incarnation, was designed by Tim Anderson, with contributions from Dave Lebling, Duncan Blanchard, and J. D. Arnold.

pdl 11/30/88

Table of Contents

Introduction to ZIP 5
ZIP Instruction Format 6
- Opcode Format 6
- Addressing Modes 7
- Instruction Values 8
- Predicates 8

ZIP Instruction Set 10
- Instruction Metasyntax 10
- Extended Opcodes 11
- Arithmetic Operations 12
- Logical Operations 14
- General Predicates 15
- Object Operations 16
- Table Operations 19
- Variable Operations 22
- Stacks 24
- I/O Operations 25
  - Input 25
  - Output 30
  - Changing Margins 33
  - Carriage Return Interrupt 33
  - Fonts and Highlighting 34
  - I/O Pseudo-devices 37
  - Windows 39
  - Pictures 43
  - Sound 44
- Control Operations 45
- Game Commands 47

ZIP Data Structures 50
- Program Structure 50
- Segment Table 56
- Global Table 57
- Object Table 58
- Vocabulary Table 59
- String Format 60
- Frequent Words 62
- Functions 63
- Picture Files 64

ZIP Opcode Summary 68
Index 71

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 have been many versions of ZIP used in developing games. These include ZIP, EZIP, XZIP, LZIP, and now, YZIP.

This document is primarily concerned with YZIP. The major differences between it and older versions of ZIP will rarely be mentioned.

ZIP 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.

ZIP Instruction Format

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.

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.

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
```

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

Single Operand (1OP)

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

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.

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 and IXCALL instructions, 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.

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

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.

ZIP Instruction Set

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 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
fcn	pointer to a function
loc	pointer to a program location

Optional arguments are indicated by 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 the same format ZAP uses).

Extended Opcodes

EXTOP opcode:int	0OP:190

This 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.

ZIL

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

This Adds the integers.

SUB arg1:int,arg2:int >VAL — 2OP:21

This Subtracts arg2 from arg1.

MUL arg1:int,arg2:int >VAL — 2OP:22

This Multiplies the integers.

DIV arg1:int,arg2:int >VAL — 2OP:23

This Divides arg1 by arg2, returning the truncated quotient.

MOD arg1:int,arg2:int >VAL — 2OP:24

This Divides arg1 by arg2, returning the remainder.

RANDOM arg:int >VAL — EXT:231

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

ument

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.

a tester might play

Arg of a negative ~~number makes~~ STET 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 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?

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 (conjunction).

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

Bitwise logical or. (disjunction).

This performs a

BCOM arg:word >VAL 1OP:248

Bitwise logical (NOT) complement.

SHIFT int,n >VAL EXT:258

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

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.

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~~ is 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?

Object Operations

Objects are the primary complex data type in ZIP. Objects are linked together in a tree structure. In addition, objects have simple (one-bit) and complex property lists.

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.

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

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 for bits 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

This Puts 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

This 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) becomes 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) becomes 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

This sets flag in obj.

FCLEAR obj,flag 2OP:12

This clears flag in obj.

LOC obj >VAL 1OP:131

This returns the "parent" of obj, zero if none.

FIRST? obj >VAL /PRED 1OP:130

This returns the "first" slot of obj. It fails if there is none (equals zero) and returns zero.

NEXT? obj >VAL /PRED 1OP:129

This returns the "next" slot of obj. It fails if there is 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

This returns the ~~specified~~ prop property of obj. If obj has no property prop, this returns the prop'th element of the default property table.

PUTP obj,prop,any EXT:227

This changes the value of obj's property prop to any. It is an 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).

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,*recspec:int* >VAL /PRED EXT:247

INTBL? is used to search for records in a table of records. In the simple case (recspec defaulted) 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.

If recspec is supplied, it is interpreted as a 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 searching a word table. Len must not be less than zero.

Note that the len argument is 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
```

Supplying the recspec 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~~ bytes from source into dest until length bytes have been copied.

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

If length is positive, this 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.

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

This Returns the value of var.

SET var,any 2OP:13

This 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.

by the calling function

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.

INC var 1OP:133

This Increments the value of var by one.

DEC var 1OP:134

This Decrements the value of var by one.

IGRTR? var,int /PRED 2OP:5

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

DLESS? var,int /PRED 2OP:4

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

Stacks

ZIP supports user stacks as well as the game stack. A user stack is defined to be a word table; instructions will check for overpush, but cannot detect overpop (the bounds-checking routines in debugging ZIPs will do that).

The 0th word of the table is the number of words now available on the stack; when it reaches 0, the stack is full, and the next push will be an overpush.

Thus a stack is just a word LTABLE; you can reset the stack by setting the 0th word to its original value.

PUSH value EXT:232

Pushes value onto the stack.

XPUSH value,stack /PRED EXT:280

The value is pushed on the stack. XPUSH returns TRUE or FALSE; if it returns FALSE, the stack was already full, and nothing was pushed.

POP *stack* >VAL EXT:233

POP takes 0 or 1 args, and pops either the game stack or the supplied stack. It returns the value popped.

FSTACK n,*stack* EXT:277

Flushes n elements from the specified stack; returns nothing. If no stack is specified, pops n things off the game stack. Only the compiler ever uses the one-argument form.

I/O Operations

Originally, ZIP was designed to perform only text input and output. As the system has evolved, capabilities have been added for windows, graphics, sound, and pointing devices.

Input

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

During the first phase of READ, 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) by ZIP 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.

The second byte of inbuf by the game is used by ZIP 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, then the second byte of inbuf should be zeroed before READ is called.

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 table pointed to by the TCHARS word. 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.

The characters typed by the user are not ~~scripted~~ put in the transcript by ZIP.

Lexv is used to store results of parsing the contents of inbuf by ZIP lexically. 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.

If the lexv argument is zero or omitted, the input accumulated by READ is not parsed. It is possible for READ to be executed (which returns 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.

[Move to ⊛ on next page.]

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. or the word is there. ←⊛

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 2 .)

These last two values are used by the program in conjunction with PRINTC to reprint words. This part of READ may be invoked separately as the LEX instruction.

READ reads text until it encounters a newline character (or any character in the TCHARS table). 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 (see the section *Vocabulary Table*). 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 (see the section *String Format*) 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 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

This 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~~ table. If supplied, it is an additional vocabulary ~~list~~ table.

*[I wish any nonzero value were returned as the value of READ/INPUT.]

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

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:int,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.

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

Function keys are accepted by both the INPUT and READ instructions.

key name	value
up-arrow	129
down-arrow	130
left-arrow	131
right-arrow	132
keys F1-F12	133-144
keypad keys 0-9	145-154
menu selection	252
mouse double click	253
mouse single click	254

The number and placement of keypad and function keys may vary from machine to 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.

MOUSE-INFO table EXT:278

The table must be four words long. It will be filled in with the current mouse status, to wit:

0	y position (in screen units)
1	x position
2	button status: one bit for each button on the mouse
3	menu/item selected.

In the button status word, bit 1 is the rightmost button, 2 is the next button, and so on. The bit will be set if the button is depressed.

In the menu/item word, the high byte is the menu number (where 1 is the default menu of SAVE/RESTORE/etc.). The low byte is the number of the item selected (the numerical offset of the selected item in the menu).

MOUSE-LIMIT window EXT:279

This restricts the mouse to a particular window. The interpreter will, if it can, not allow the mouse cursor outside the specified window (not all machines support this). Mouse events will not be reported unless the mouse cursor is inside the specified window. Giving a window argument of -1 removes the constraints. Initially, the mouse should be assumed to be constrained inside window 1.

The MOUSE-INFO instruction will report the mouse position even if it's outside the specified window.

Note that moving/resizing the window will move/resize the allowed mouse area.

MENU id,tbl /PRED EXT:283

The purpose of this is to add a menu to the menu bar. (The Macintosh is the only machine that currently supports this instruction).

If the MENU capability bit is not set in the FLAGS word, MENU returns FALSE.

The id argument is an integer greater than 2, specifying which slot in the menu bar is filled. On the Macintosh, slot 0 is the apple menu; slot 1 is the File menu (Save, Restore, etc.); slot 2 is the Edit menu. All three are reserved. The tbl is an LTABLE of character LTABLEs. The first element is the menu name, which will actually appear in the menu bar; the remaining elements are the menu elements. When a menu element is selected, the INPUT/READ instruction will return 252, whereupon a MOUSE-INFO will reveal the selected element. To take a menu down, MENU id,0.

MENU returns FALSE if there's no room in the menu bar for the requested menu.

[Change "print..." to "display..." throughout? Or add a comment.]

Output

ZIP allows a screen of any width between 60 and 80 columns, and any height from about 14 lines on up.

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 a scrolling window (such as window 0) has filled the text area, a [MORE] prompt will be printed. A character will be read from the ~~terminal~~ keyboard before additional output is printed.

BUFOUT int EXT:242

[This should be moved down, like to p.36.]

This 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.

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

Use of BUFOUT is rarely 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, and DIROUT are implemented in a particular interpreter, they may also want to output buffered text.

HLIGHT and FONT present special problems. In general, one only wants carriage-returns to happen at "word breaks," such as space. If HLIGHT (for example) outputs the buffer, characters with a different highlighting may end up on the next line even if they would not have if though

output with normal highlighting. One way to avoid this problem is for HLIGHT and FONT to output special marker characters into the output buffer, and then perform the actual highlight or font changing ~~when~~ as these characters are encountered ~~then~~ when the buffer is actually output to the screen.

PRINTC int EXT:229

This Prints the character whose ASCII value is int.

PRINTN int EXT:230

This Prints int as a signed number.

PRINT str 1OP:141

This Prints the string pointed to by str times four plus the contents of SOFF times eight. The calculation is necessary because str in this instruction is a quad-pointer to ~~the~~ offset ~~of the string~~ in the string area, guaranteed to point to a string that has been quad-aligned.

PRINTB str 1OP:135

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

PRINTD obj 1OP:138

This Prints the short description (DESC property) of obj.

PRINTI (in-line string) 0OP:178

This Prints an immediate string. This is 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

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

CRLF 0OP:187

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

PRINTT bytes:tbl,width:int,*height:int,skip:int* EXT:254

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.

PRINTF tbl EXT:282

A new instruction which prints a "formatted" table. It takes only a pointer to the table, because all the other information is stored in the table. It is expected that this sort of table will normally be generated by DIROUT 3,tbl.

PRINTF acts much like PRINTT, in that it prints a rectangular block of text, but unlike PRINTT, it allows the lines to be different lengths when stored. This can happen due to use of variable width fonts, imbedded highlighting characters, etc.

A formatted table looks like:

```
number-of-characters-in-line-1:word
character-1:byte
character-2:byte
character-n:byte
number-of-characters-in-line-2:word
 ...
number-of-characters-in-line-n:word
 ...
0:word
```

The zero count at the end signifies the end of the table.

Changing Margins

MARGIN left:int,right:int,*window* EXT:264

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.

On a non-wrapping window, MARGIN is a no-op. MARGIN 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 a window should position the cursor at the left margin of the top line of the window.

Carriage Return Interrupt

Two window attribute 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.

by [?]

Fonts and Highlighting

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:

mode	interpretation
0	no highlight
1	inverse video
2	bold
4	underline or italic at the interpreter's discretion
8	monospaced font

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,*window* >VAL EXT:260

This selects a particular font for the specified 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~~ is used for all new text in 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~~ all screen windows. The interpreter is responsible for updating the FWRD parameter word whenever the font changes.

Known fonts include:

num	font
0	previous font
1	normal font
2	picture font -- obsolete?
3	VT100 character graphics font
4	monospace font

FONT prints and empties the output buffer.

Note that unlike the monospace highlighting mode, FONT 4 may be combined with various highlighting modes to produce (for example) bold monospace output.

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

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

If MODE bit 0 (XCOLOR) is zero, this operation is ignored.

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:

value	color
-1	color of pixel at cursor position
0	no change
1	system default color
2	black
3	red
4	green
5	yellow
6	blue
7	magenta
8	cyan
9	white
10	light gray (Amiga only)
11	gray (Amiga only)
12	dark gray (Amiga only)

Several caveats apply to the use of the COLOR opcode. First, even on machines that support color selection, the colors available may vary.

For example, the AMIGA allows several shades of gray to be selected, as the standard white is 'too white.' In addition, the colors selected on the AMIGA for the text window (window 0) ~~become global on~~ are used in all windows.

Games should allow for machines (Apple IIs, monochrome Macs, etc.) that don't permit color selection. Machines with monochrome display as an option should ask the player whether he or she wants color.

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.

USL 0OP:188

This instruction is obsolete and should not be used.

I/O Pseudo-devices

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

This 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. Each known device will be discussed separately.

DIROUT 1

directs output to the screen. This is the initial state. The screen may be shut off by DIROUT -1. This is useful, for example, to send text to the transcript device without it appearing on the screen.

DIROUT 2

directs output to the transcript device. It sends a transcript of ~~player input and~~ all output in any window with scripting enabled to a transcript, which may be a file, a printer, or any appropriate device. Transcripting is terminated when DIROUT -2 is performed. 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 ~~window output~~ player input to the transcript.

(The existence of a resumable READ instruction implies that the input buffer ~~read by READ~~ must be output ~~manually~~, or the script file ~~will~~ would end up with a copy of the input buffer each time READ returns.)

DIROUT 3,tbl,just

directs output to the table output device. Output is sent to the table specified as tbl.

Each character printed when table output is enabled is PUTB'd into the tbl starting at the table beginning plus two bytes. When a DIROUT -3 is performed, the number of characters printed is PUT into the tbl at offset 0. If the just argument is not supplied, DIROUT also keeps track of the width of the characters output to the table in pixels. The lowcore location TWID is initialized to zero when DIROUT 3 is executed, and each time a character is output, its width in pixels is

added to TWID. This enables the programmer to (for example) do right-justified text in variable width fonts.

DIROUT 3 is more complex if the just argument is supplied. In that case, just is either a window, or the negative of a width in pixels. Output to the table should be justified as though it was being sent to the window or to a window just pixels wide. In either case, the result is a "formatted" table, suitable for passing to PRINTF. The way this is accomplished is to wrap output at the width given, padding it if necessary. Each time a line is filled, the number of characters output is PUT in offset 0 of the table's last line, the count is reset to zero, and DIROUT skips a word to use as the byte count for the next line. When a DIROUT -3 is performed, the last line is padded, its count is PUT, and a word of zero (meaning an empty line) is PUT.

When a carriage-return line-feed is printed, a 13 (hex $0D) is placed in the table, unless the table is to be a formatted one. In the formatted case, the line is padded with spaces to fill it out.

Output redirected to a table is not buffered. When the table device is selected, all other devices are ignored until it is deselected.

DIROUT 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.

DIRIN 0

is the keyboard (this is the default case).

DIRIN 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.

Windows

ZIP provides eight windows, numbered 0 through 7. The programmer may set each window's position, size, and characteristics, using the following operations. Note that this is not a window system where one creates new windows; all the windows exist all the time. In fact, they are more like "parts of the screen" than windows in the Mac sense. Each window has many attributes, stored by ZIP and readable and settable through special instructions described below.

[Use bold or some other highlight.]

wrap/not wrap: whether text should be wrapped or clipped when it would extend past the right margin

scroll/not scroll: whether the window should scroll when a carriage return is printed at the bottom

script/not script: whether output in the window should be scripted. Scripting is still controlled by the scripting bit in FLAGS; this bit therefore means "script output in this window if scripting is enabled."

buffered/unbuffered: whether text output to this window should be buffered or not.

Windows are defined to be transparent; although wrapping or clipping will occur on the edges of a window, it will not occur if another window overlays part of the current window. Similarly, moving/resizing a window doesn't conceal or reveal anything.

Any operation that can take a window as an argument will interpret an argument of -3 to mean "the current window." (It's -3 to avoid a collision with the range of arguments to CLEAR.)

In the initial configuration, window 0 occupies the whole screen; it scrolls, wraps, and scripts. Window 1 occupies the full width of the screen, but has 0 vertical dimension. All other windows have size 0 in both dimension, live at 1,1, and have none of the scrolling, scripting, wrapping (or any subsequently defined) attributes. All windows initially have buffering on.

SCREEN window:int	EXT:235

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

When a window is departed, the cursor position in that window is remembered, and when the window is reentered, it is restored.

WINPOS window:int,y,x EXT:272

Sets the location of the top left corner of the specified window, relative to the top left corner of the screen. The coordinates are 1-based, as for CURGET/CURSET, and are in screen units (thus, a window whose origin is at the origin has WINPOS 1,1). The window's saved attributes, including cursor position, are unchanged; there is no visible effect unless this is the current window, in which case the cursor will move to its same position, relative to the new window location, as it was in before the WINPOS.

Note that while this instruction should check for a window position that is offscreen, it should not check for a window size that is too big, given the new window position.

WINSIZE window,y,x EXT:273

Sets the size of the window, in screen units. If the cursor is outside the new area, it is moved to 1,1. There is otherwise no visible effect.

Note that while this instruction should check for a window size that won't fit the physical screen, it should not check for a window position that is out of range given the new window size.

SPLIT height:int EXT:234

The SPLIT operation affects only the vertical dimensions of windows 0 and 1, and the vertical position of windows 0 and 1. For example, SPLIT 2 sets the vertical position of window 1 to 1, and its vertical size to 2. The vertical position of window 0 becomes 3 and its vertical size becomes old_size - (old_top - 3), or 0 if that quantity is negative. Thus, the bottom of window 0 is not affected. After this operation, window 0 is selected.

SPLIT 0 sets the vertical size of window 1 to 0, moves the top of window 0 to the top of the screen, and selects window 0. It has no other effect.

Use of SPLIT is discouraged. Instead, use appropriate invocations of WINPOS and WINSIZE.

WINATTR window,bits,*operation* EXT:274

Sets new characteristics of the window, currently wrapping (1), scrolling (2), scripting (4), and buffering (8). The operation, if not supplied, is MOVE, meaning change all of the window's characteristics according to the bits argument. Operations include: MOVE (op 0)

changes all the attributes to new values, SET (op 1) sets any attributes that are on in ~~the~~ bits ~~arg~~., CLEAR (op 2) clears any etc., COMP (op 3) complements any ~~attributes whose bits are on in the bits arg.~~ etc.

CLEAR window:int EXT:237

If window is 0 to 7, CLEAR clears window #window. If window is -1, it unsplits the screen (if it has been split) and clears the entire screen. CLEAR -2 just clears the whole screen, and has no effect on window attributes, including cursor position.

When a window is cleared, the cursor moves to the top and left of that window. The window is cleared to the current background color.

ERASE int EXT:238

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.

If int is greater than 1, it erases an area int pixels wide and the font height high, starting at the cursor position. In no event will it erase past the right edge of the current window.

The ERASEd area ~~should be~~ is colored the background color.

WINGET window,offset >VAL EXT:275

This Returns the window's value for the property numbered offset. Legal property numbers are:

name	offset	interpretation
WTOP	0	y position
WLEFT	1	x position
WHIGH	2	y size
WWIDE	3	x size
WYPOS	4	y cursor position
WXPOS	5	x cursor position
WLMARG	6	left margin
WRMARG	7	right margin
WCRFCN	8	carriage return interrupt function
WCRCNT	9	carriage return interrupt counter
WHLIGHT	10	highlight mode
WCOLOR	11	color word (background, foreground)
WFONT	12	font ID
WFSIZE	13	font size (height, width)
WATTRS	14	attributes
WLCNT	15	line counter ← [What's this?]

WINPUT window,offset,value EXT:281

This is Exactly like WINGET except that the value is written to the window property table instead of read from it. Note, however, that most window properties are only writeable with special instructions (such as WINSIZE, etc.)

SCROLL window,*lines* EXT:276

This Scrolls the specified window up or down by lines (default 1) pixels. The cursor position is not affected. Positive arguments scroll up, negative arguments scroll down. Blank lines are inserted in the background color of the specified window. This works whether the window is a scrolling window or not.

CURSET y:int,*x:int,window* EXT:239

The cursor position in (immediate) the current window (if window is not supplied) or the specified window is set to y,x. If the specified window is not current, there is no visible effect. Note that y and x are relative to the top-left corner of the specified window, and that this is defined to work in any window. If either of the arguments is outside the area covered by the window, it will be set to the appropriate dimension of the window.

The upper left corner of a window is the origin, and is referred to as 1,1.

CURSET ~~must~~ output any buffered output before actually moving the cursor.

If CURSET is given a y argument of -1, it means "turn off the cursor." An argument of -2 means "turn on the cursor." In this case, CURSET may take a single argument. This facility is provided for machines that have unsightly cursors that want to be out of sight at certain times.

CURGET output:tbl EXT:240

This 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.

In general, it is preferable to use WINGET for this operation.

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,y:int,x:int EXT:261

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

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

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

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 *Picture Files* for an example specification of how pictures are stored.

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

This clears the area taken up by the picture, i.e., restores the window background color.

Sound

SOUND id:int,*op:int,volume:int,repeat:int* EXT:245

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

SOUND takes a sound-identifier argument, and a sound-operation argument as well. Currently, there are only ~~three~~ these 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~~ or "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.

If the third argument, volume is supplied, it sets the volume at which the sound is to be played. -1 is the default volume. If the fourth argument, repeat is supplied, it is a count of how many times to play the sound. -1 means to play the sound until it is explicitly stopped.

Control Operations

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

This Begins execution of the function (see the section *Functions*) pointed to by fcn times four plus FOFF times 8, supplying it with any arguments given in the CALL instruction. Note that fcn is a quad-pointer and the first location in a function is 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
CALL2 fcn,any >VAL	2OP:25
XCALL fcn,any1,any2,any3,any4,any5,any6,any7 >VAL	EXT:236

These are the same as CALL, but use the more compact instruction coding formats where possible. They are never explicitly invoked by the programmer, but are generated by the compiler instead.

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

These 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 opcodes 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. (S)

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

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

RTRUE 0OP:176

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

RFALSE 0OP:177

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

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

CATCH returns a pointer (called a frame) to the current 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.

JUMP loc 1OP:140

This makes 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 two's-complement word.

RSTACK 0OP:184

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

NOOP 0OP:180

This is No operation, equivalent to a "JUMP 2."

Game Commands

SAVE *start:tbl,length:int,name:tbl* >VAL EXT:256

If given no arguments, this 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. The state of windows and other hardware dependent information is not part of a SAVE file, as these might need to be recomputed due to different screen size, or different capabilities of a particular machine.

Details of the user interface are left to the discretion of the implementor.

When given three arguments, SAVE (and RESTORE) may be used as atomic i/o operations, as follows:

SAVE writes a section of the impure area beginning at start. 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.

In high-level terms, the game saves or restores a TABLE or a group of contiguous tables.

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

RESTORE *start:tbl,length:int,name:tbl* >VAL EXT:257

If given no arguments, this 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.

RESTORE does not restore the state of windows, colors, fonts, etc., as these are not part of the SAVE file. After doing a RESTORE, games should re-perform any special initialization, or reset any window parameters that might be different in the SAVE file.

With three arguments, RESTORE is an atomic i/o operation. STET ~~RESTORE reads~~ a section of the impure area.

See SAVE for details about the ~~other~~ arguments. No A

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 the 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, (i.e. during this session) IRESTORE should return 0.

VERIFY /PRED 0OP:189

This 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 0OP:191

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

RESTART 0OP:183

this Reinitializes the game, reloads the preload area from disk, and generally acts as if it had just been started.

QUIT 0OP:186

The game should die peacefully.

ZIP Data Structures

Program Structure

A Z-language program begins with the following words (those underlined 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	points to beginning of non-preloaded code
3	START	points to function where execution begins
4	VOCAB	points to vocabulary table
5	OBJECT	points to object table
6	GLOBALS	points to global variable table
7	PURBOT	points to beginning of pure code and interpreter-settable
8	FLAGS	16 game-settable flags
9	SERIAL	serial number - 6 bytes
12	FWORDS	points to fwords table frequent
13	PLENTH	length of program right-shifted by 3
14	PCHKSM	checksum of all bytes
15	INTWRD	interpreter identification word
16	SCRWRD	screen parameters word
17	HWRD	width of display in pixels
18	VWRD	height of display in pixels
19	FWRD	one byte font height, one font width
20	FOFF	points to start of function area
21	SOFF	points to start of string area
22	CLRWRD	one byte background color, one foreground
23	TCHARS	pointer to table of terminating characters
24	TWID	output location for DIROUT
25		unused
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 extension table words have been defined but no ZIP implements them)

BUTTON	points to button handler
JOYSTICK	points to joystick handler
BSTAT	button status
JSTAT	joystick status

ZVERSION is interpreted as two bytes, VERSION and MODE. All games produced in YZIP will have a Z-machine version byte of 6; XZIP games will have a 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.

bit #	name	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	

these bits are set by the compiler for a particular game and may be changed by
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 (576K) [?] 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 This 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	%FSOUN	game uses sounds
8	%FMENU	game uses menus
9-15	reserved	

Bit #1 (%FFIXE, EZIP only) ~~should~~ will 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~~ will be set by the interpreter whenever, in its opinion, the ~~status line area~~ screen 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)~~ screen 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~~ want to use 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~~ does not support graphics mode, it clears this bit at initialization time. in this session

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. The interpreter clears this bit if ISAVE + IRESTORE are not available.

Bit #5 (%FMOUS) should be set by games which want to use the mouse. The bit ~~should~~ will be cleared by the interpreter if no mouse is available.

Bit #6 (%FCOLO) should be set by games which ~~will be using~~ want to use 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.

Bit #7 (%FSOUN) should be set by games that wish to use sounds. The bit ~~should~~ will be cleared by the interpreter if it does not support SOUND.

Bit #8 (%FMENU) should be set by games that wish to use menus. The bit ~~should~~ will be cleared by the interpreter if it does not support MENU.

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~~ that the game was ~~compiled~~ assembled.

PLENTH and PCHKSM are both used by the VERIFY operation. PLENTH is the length of the game, in bytes, divided by eight. PCHKSM is the 16-bit sum of all bytes from 64 (decimal) to PLENTH*8-1.

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 #
Debugging Int.	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, a number which identifies the release of the given interpreter. A particular interpreter is usually referred to by the combination of VERSION and INTVR, for example, "Amiga interpreter 6.3". 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).

FOFF is the offset of the function area of the game. This value is stored as an oct-~~word~~ address. That is, the actual address is the contents of FOFF shifted left three bits. Any function address, for example, the first argument to a CALL type instruction, is converted to its actual address by shifting it left two bits (recall that functions are quad-byte aligned) and then adding FOFF shifted left three bits. This allows a game to have nearly 256K of functions. It is okay for FOFF to be zero.

SOFF is the offset of the string area of the game. This value is stored as an oct-~~word~~ address. That is, the actual address is the contents of SOFF shifted left 3 bits. Any string address, for example, the first argument to the PRINT instruction, is converted to its actual address by shifting it left two bits (recall that ~~functions~~ are quad-byte aligned) and then adding SOFF shifted left three bits. This allows a game to have nearly 256K of string space. It is okay for SOFF to contain zero.

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.

TWID is used by DIROUT to a table to record the width of the characters output, in pixels. It is updated each time a character is output, and zeroed when DIROUT 3 is performed.

CHRSET is used to define the printing character set. 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, and the first two bytes of the third group of 26 bytes are ignored, as they would represent the ascii escape and the in-line carriage-return.

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.

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.

Segment Table

Between pictures and increased game size, ZIP must support games that take more than a single disk on some systems.

The simple approach (which might be adequate for IBM 360K disks) puts the game file on one disk and the picture file on another; the user should have two floppy disk drives (or a hard drive) to play without a lot of disk-swapping.

For the Apple II, however, even this isn't adequate: one side of a disk (the drives can only read one side) holds 140K, and the interpreter can preload about 55K, so any game over 195K has to go to more than one swapping surface.

The first thing in the game file on machines that support multiple disks will be a segment table. The format is (all quantities are 16 bits):

number of words following
number of disks
  for each disk
  number of segments
    for each segment
    starting virtual page #
    ending virtual page #
    starting disk page #

This table is only visible to the interpreter. Its use is: Given an address (after any shifting and relocation; we want the actual virtual address, rather than the mere function offset), it must first be checked to see whether it's in ~~core~~ RAM, either because it's preloaded or because its page is currently swapped in. If it isn't, the interpreter first checks the segment table for the current disk to see if there's a segment containing the desired page on the current disk; if so, the page is swapped in. If not, the interpreter scans the segment tables for all the disks until it gets a match; it then asks the user to insert that disk (it may first check ~~the current disk or~~ the other floppy drive for it), and begins swapping from it.

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.

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.]

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

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~~ of 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.

String Format

For maximum storage efficiency, text is encoded in 5-bit bytes called 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.

value	meaning
0	space
1	frequent word 0-31 (determined by zbyte following)
2	frequent word 32-63 (determined by zbyte following)
3	frequent word 64-95 (determined by zbyte following)
4	shift character set (effect varies depending on current set)
5	shift character set (effect varies depending on current set)

Values 4 and 5 have different effects depending on what the current character set is. 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

Once these special values are out of the way, 76 characters may be represented "compactly." The CHRSET word in low core points to a table 78 bytes long (two bytes in character set two are zero) which defines which characters get this special treatment. CHRSET need not be supplied, as ZIP has an initial definition of the "compact" character set.

Initial Character Sets

character set value

set	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
2	*	*	0	1	2	3	4	5	6	7	8	9	.	,	!	?	_	#	'	"	/	\	-	:	(	)

*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). These two values are special and do not change with CHRSET. A CHRSET table will contain two zeros in the positions occupied by these two values.

Characters are always output as they appear in the ~~print~~ string, with two exceptions. The TAB character (Control-I), if it appears at the beginning of a line, is output as a "suitable" amount of paragraph indentation for the machine and font being used. Otherwise, TAB is equivalent to space. The Control-K character is output as a "suitable" sentence break for the machine and font being used.

At the beginning of each string, the initial permanent character set is ~~should be~~ 0, with no temporary mode selected. The encoding algorithm used to create the string determines under what circumstances a permanent shift rather than a temporary one is to be used.

Frequent Words

The frequent words table, pointed to by FWORDS and below ENDLOD (a pure table), contains 96 quad-pointers to ordinary strings. Note that unlike normal string pointers, these are not offset by SOFF. These strings represent frequently used substrings (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.

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 or IXCALL instruction).

Functions are referred to by a pointer to the absolute quad-byte address of the function minus the function offset. The function offset is the contents of FOFF left shifted three bits.

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).

Locals are initialized to zero. 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~~ will be tested with ASSIGNED? (q.v.) to determine whether to use the default values.

The ~~value words are~~ count of locals is 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 function 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~~ function, and whether the calling ~~routine~~ function is expected to return a value.

Picture Files

Rather than storing each picture for a game in a separate file, ZIP stores them in a single file (or at least a small number of files). In addition to saving the file overhead, this facilitates sharing of palettes and Huffman coding information.

A given picture file for a game is identified by a small number (less than 256); the number of picture files actually used will depend on disk size (a Mac might only need one) and number of pictures. We do not require that a given picture occur in only one file; in a multi-disk game, a picture might have to exist on more than one of the disks to avoid disk-swapping.

A picture file consists of three or four parts: a header, a local directory, an optional global directory, and data. In order:

Header:

File ID (1 byte)
Flags (1 byte)
Pointer to global Huffman tree (2 bytes)
Number of entries in local directory (2 bytes)
Number of entries in global directory (2 bytes)
Directory entry size, in bytes (1 byte)
Extra (7 bytes) (total 16 bytes)

Current flags are:

1 This file has a global directory.
2 Some pictures in this file are Huffman-coded, so all local directory entries will have a pointer to a Huffman tree (it may be 0), depending on the setting of the 4 bit.
4 All pictures in this file are Huffman-coded together, so there is only one tree stored. The pointer to it is the next word in the header; to get the actual file offset, multiply the stored pointer by two.
8 No palette information is stored with the pictures in this file. This applies mostly to the Apple, where all pictures have the same palette anyway. In this case, there's a slightly different directory format.

The global Huffman tree pointer is non-zero only if the 4 bit is set in the flags byte.

Local directory:

The number of entries and the size of each entry are specified in the file header.

Each entry has the following format:

Picture ID (16 bits)
Picture X size (16 bits)
Picture Y size (16 bits)
Flags (16 bits)
Pointer to picture data (24 bits)
Pad byte (optional)
Pointer to palette (24 bits) (optional)
Pointer to Huffman tree (optional) (16 bits)

The smallest entry is on the Apple II, at 12 bytes -- it has neither the palette pointer nor the Huffman information. On other machines, the Huffman information may be missing, giving an entry size of 14 bytes; with everything present, the entry size is 16 bytes.

The palette and picture data pointers are simply absolute offsets into the picture file. The Huffman tree pointer is doubled before use, meaning that it's really 17 bits wide. This means that all Huffman trees must live in the first 128K of the file. The format of the data is described below.

Global directory:
The global picture directory, if present, is used to find the file containing a picture that's not in the current file. In a game with only one picture file, it isn't necessary; with care, it probably isn't needed in a multi-disk game either. But we'll describe it anyway.

The number of entries is contained in the file header. Each entry is simply:

Picture ID (16 bits)
File ID (16 bits)

The directory is sorted by picture ID. It contains entries only for pictures that are NOT in the current file, rather than entries for all pictures used by the game.

If a group of pictures is referenced in a PICSET operation, then ALL pictures in the group must be in the same file(s).

Data:
The data area consists of raw bits, representing Huffman trees, palettes, and picture data.

A Huffman tree (which must start on an even byte boundary) has no header, because none is needed. The maximum size is 256 bytes. Each node of the tree is represented by a pair of bytes; the root node is the pair at the location referenced by the directory entry. The

structure is quite simple: the high-order byte is the child for a 0 bit; the low-order byte is the child for a 1 bit. If the byte value is greater than 127, then the child is a terminal node, representing x-128. Otherwise, the child is a nonterminal; node number x is at location 2x from the beginning of the tree.

A palette has two parts: the color palette, and the stipple palette.

The color palette comes first:

number of entries (1 byte)

r (1 byte) g (1 byte) b (1 byte) for each entry. If the number of entries is 14 or fewer, the first color number actually used in the picture is 2; if 15, the first color number used is 1. It would be possible for the number of entries to be 0, if the data file is for a machine that doesn't support color.

The stipple palette simply immediately follows the color palette:

number of entries (1 byte) (always either 0 or 16)

stipple IDs

The stipple palette has the following interpretation: we assume a set of 16 stipples, numbered 0 through 15. Stipple 0 is all black; stipple 15 is all white. Stipples 1 through 4 are 1/4 white; 11 through 14 are 3/4 white; 5 through 10 are 1/2 white. A given color number is used to index into the stipple palette to determine which stipple to use in displaying that color.

The number of entries can be 0 on a machine where stippling will never happen; the apple II comes immediately to mind (and possibly the Mac II, where a grey scale is available).

The data for a picture must be interpreted according to the flags field of its directory entry, but the basic format is simply 16 bits of size, followed by raw data. The following flags are currently defined:

1 This picture has a transparent section; the color number of the transparency is specified by the high four bits of the flags word. (These will normally be 0 except on the Apple II.)
2 This picture is Huffman coded.
4 This picture was XORed on alternate rather than adjacent lines.
8 This is a two-color picture: color 2 is white, color 3 is black. Color 0 is still transparent if bit 1 is set.

If the picture is Huffman-coded, the first two bytes of `raw data' are the length of the data before it was Huffman-coded. This is required to avoid using bad bits at the end of the last byte of Huffman data.

The data that results after the Huffman decoding (or that is just in the picture file, if Huffman coding didn't happen) is compressed in the following way (the decompression just happens in reverse order): first, each line is XORed with the line above it (if the 4 bit is set in flags, with the line two above it). At this point the picture consists of bytes, many of them 0 due to the XOR. The run-length encoding is applied as follows:

or

`Runs' one or two long aren't changed. Longer runs are encoded as:

byte value ? count where the count is 15 + run_length -1.

Thus, a run of three 0's becomes 0 ? 17. If the picture was ~~Huffed,~~ Huffman coded the count byte will always be less than 128; otherwise, it may go as high as 255. Note that the run-length encoding ignores line boundaries.

ZIP Opcode Summary

What follows is an alphabetically arranged list of all ZIP opcodes.

ADD arg1:int,arg2:int >VAL	2OP:20
ASHIFT int,n >VAL	EXT:259
ASSIGNED? opt:var /PRED	EXT:255
BAND arg1:word,arg2:word >VAL	2OP:9
BCOM arg:word >VAL	1OP:248
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
CLEAR window:int	EXT:237
COLOR fore:int,back:int	2OP:27
COPYT source:tbl,dest:tbl,length:int	EXT:253
CRLF	0OP:187
CURGET output:tbl	EXT:240
CURSET y:int,x:int,*window*	EXT:239
DCLEAR picture:int,y:int,x:int	EXT:263
DEC var	1OP:134
DIRIN device:int,*any1*,*any2*,*any3*	EXT:244
DIROUT device:int,*any1*,*any2*,*any3*	EXT:243
DISPLAY picture:int,y:int,x:int	EXT:261
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
FCLEAR obj,flag	2OP:12
FIRST? obj >VAL /PRED	1OP:130
FONT font:int,*window* >VAL	EXT:260
FSET obj,flag	2OP:11
FSET? obj,flag /PRED	2OP:10
FSTACK n,stack	EXT:277
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
ICALL1 routine:fcn	1OP:143
ICALL2 routine:fcn,arg1:any	2OP:26

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,*recspec:int* >VAL /PRED	EXT:247
IRESTORE >VAL	EXT:266
ISAVE >VAL	EXT:265
IXCALL routine:fcn,arg1,...	EXT:250
JUMP loc	1OP:140
LESS? arg1:int,arg2:int /PRED	2OP:2
LEX inbuf:tbl,lexv:tbl,*lexicon:tbl,preserve:bool*	EXT:251
LOC obj >VAL	1OP:131
MARGIN left:int,right:int,*window*	EXT:264
MOD arg1:int,arg2:int >VAL	2OP:24
MOUSE-INFO table	EXT:278
MOUSE-LIMIT window	EXT:279
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
PICINF picture:int,data:tbl /PRED	EXT:262
POP *stack* >VAL	EXT:233
PRINT str	1OP:141
PRINTB str	1OP:135
PRINTC int	EXT:229
PRINTD obj	1OP:138
PRINTF tbl	EXT:282
PRINTI (in-line string)	0OP:178
PRINTN int	EXT:230
PRINTR (in-line string)	0OP:179
PRINTT bytes:tbl,width:int,*height:int*	EXT:254
PTSIZE table >VAL	1OP:132
PUSH value	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
REMOVE arg:obj	1OP:137
RESTART	0OP:183
RESTORE *start:int,length:int,name:tbl* >VAL	EXT:257
RETURN any	1OP:139
RFALSE	0OP:177
RSTACK	0OP:184
RTRUE	0OP:176
SAVE *start:int,length:int,name:tbl* >VAL	EXT:256
SCREEN window:int	EXT:235

SCROLL window,*lines*	EXT:276
SET var,any	2OP:13
SHIFT int,n >VAL	EXT:258
SOUND id:int,*op:int,volume:int,repeat:int*	EXT:245
SPLIT height:int	EXT:234
THROW any,frame	2OP:28
USL	0OP:138
VALUE var >VAL	1OP:142
VERIFY /PRED	0OP:189
WINATTR window,bits,*operation*	EXT:274
WINGET window,offset >VAL	EXT:275
WINPOS window:int,y,x	EXT:272
WINPUT window,offset,value	EXT:281
WINSIZE window,y,x	EXT:273
XCALL fcn,any1,any2,any3,any4,*any5,any6,any7* >VAL	EXT:236
ZERO? arg:any /PRED	1OP:128
XPUSH value,stack /PRED	EXT:280
ZWSTR inbuf:tbl,inlen:int,inbeg:int,zword:tbl	EXT:252

[Some dots from left column to right would help.]

Index

Instructions
ADD **12**
ASHIFT **14**
ASSIGNED? **22**
BAND **14**
BCOM **14**
BOR **14**
BTST **14**
BUFOUT **30**
CALL **45**
CALL1 **45**
CALL2 **45**
CATCH **46**
CLEAR **41**
COLOR **35**
COPYT **21**
CRLF **31**
CURGET **42**
CURSET **42**
DCLEAR **43**
DEC **22**
DIRIN **38**
DIROUT **37**
DISPLAY **43**
DIV **12**
DLESS? **23**
EQUAL? **15**
ERASE **41**
EXTOP **11**
FCLEAR **17**
FIRST? **17**
FONT **34**
FSET **17**
FSET? **17**
FSTACK **24**
GET **19**
GETB **19**
GETP **17**
GETPT **20**
GRTR? **13**
HLIGHT **34**
ICALL **45**
ICALL1 **45**
ICALL2 **45**
IGRTR? **23**
IN? **17**
INC **22**
INPUT **27**
INTBL? **20**
IRESTORE **48**
ISAVE **48**
IXCALL **45**
JUMP **46**
LESS? **13**
LEX **26**
LOC **17**
MARGIN **33**
MENU **28**
MOD **12**
MOUSE-INFO **28**
MOUSE-LIMIT **28**
MOVE **16**
MUL **12**
NEXT? **17**
NEXTP **18**
NOOP **46**
ORIGINAL? **48**
PICINF **43**
POP **24**
PRINT **31**
PRINTB **31**
PRINTC **31**
PRINTD **31**
PRINTF **32**
PRINTI **31**
PRINTN **31**
PRINTR **31**
PRINTT **32**
PTSIZE **20**
PUSH **24**
PUT **19**
PUTB **19**
PUTP **17**
QUIT **49**
RANDOM **12**
READ **25**
REMOVE **16**
RESTART **49**
RESTORE **47**
RETURN **45**

RFALSE **46**
RSTACK **46**
RTRUE **46**
SAVE **47**
SCREEN **39**
SCROLL **42**
SET **22**
SHIFT **14**
SOUND **44**
SPLIT **40**
SUB **12**
THROW **46**
USL **36**
VALUE **22**
VERIFY **48**
WINATTR **40**
WINGET **41**
WINPOS **40**
WINPUT **42**
WINSIZE **40**
XCALL **45**
XPUSH **24**
ZERO? **15**
ZWSTR **27**
Lowcore bytes
ENDLOD 62
FNTH **53**
FNTV **53**
INTID **53**
INTVR **53**
MODE **51**, **52**
SCRH **53**
SCRV **53**
VERSION **51**, 53
Lowcore words
BSTAT **55**
BUTTON **55**
CHRSET **54**, 60
CLRWRD 35, **54**
ENDLOD **51**
EXTAB **54**
FLAGS 28, 37, 39, **51**
FOFF 45, **54**
FWORDS 62
FWRD **53**
HWRD **53**
INTWRD **53**
JOYSTICK **55**
JSTAT **55**
MSLOCX **54**
MSLOCY **54**
PCHKSM 53
PLENTH **53**
SCRWRD **53**
SERIAL **53**
SOFF 31, **54**
TCHARS 25, 26, **54**
TWID 37, **54**
VWRD **53**
ZORKID **51**
ZVERSION **51**