ZIL Reference Guide

Introduction

There are two classe of commands.

The first class is things that only work outside a "routine". These commands is processed during compilation and are a subset of MDL. The order is important and things needs to be declared before (higher up in the file) before they are used.

The second class is things that only work inside "routines". These commands is processed by til Z-machine during runtime. The order these are organized does not matter.

Sources:

Learning ZIL, Steve E. Meretzky

ZIL Course, Marc S. Blank

Syntax

Typename	**Size**	**Min-Max**	**Examples**

FIX	32-bit signed integer	-2147483648 to 2147483648	`616`
			`*747*`
			`#2 10110111`

CHARACTER	8-bit	0 to 255	`!\A`
BYTE	8-bit	0 to 255	`65`

=== MDL SUBRs and FSUBRs ===

(i.e. "things you can use outside a routine")

The syntax for most of these commands are much like the syntax in MDL.

All these commands is possible to run, test and debug during the interactive mode of ZILF (start ZILF without any options).

Sources:

The MDL Programming Language, S. W. Galley and Greg Pfister

ZIL Language Guide, Jesse McGrew

* (multiply)

```
<* numbers ...>
```

Multiply `numbers`.

Example:

```
<* 2 3 4> -->  24
```

+ (add)

```
<+ numbers ...>
```

Add `numbers`.

Example:

```
<+ 2 3 4> -->  7
```

- (subtract)

```
<- numbers ...>
```

Subtract first `number` by subsequent `numbers`.

Example:

```
<* 8 3 4> -->  1
```

/ (divide)

```
</ numbers ...>
```

Divide first `number` by subsequent `numbers`.

Example:

```
<* 20 5 2>     -->  2
```

0?

```
<0? value>
```

Predicate. True if `value` is 0 otherwise false.

1?

```
<1? value>
```

Predicate. True if `value` is 1 otherwise false.

==?

```
<==? value1 value2>
```

=?

```
<=? value1 value2>
```

ADD-TELL-TOKENS

```
<ADD-TELL-TOKENS {pattern form} ...> **F
```

ADD-WORD

```
<ADD-WORD atom-or-string [part-of-speech] [value] [flags]>
```

ADJ-SYNONYM

```
<ADJ-SYNONYM original synonyms ...>
```

AGAIN

```
<AGAIN [activation]>
```

ALLTYPES

```
<ALLTYPES>
```

returns a VECTOR containing just those ATOMs which can currently be returned by TYPE or PRIMTYPE.

AND

```
<AND conditions ...> **F
```

AND?

```
<AND? Values ...>
```

ANDB

```
<ANDB numbers ...>
```

APPLICABLE?

```
<APPLICABLE? Value>
```

APPLY

```
<APPLY applicable args ...>
```

APPLYTYPE

```
<APPLYTYPE atom [handler]>
```

ASCII

```
<ASCII {number | character}>
```

ASSIGNED?

```
<ASSIGNED? atom [environment]>
```

ASSOCIATIONS

```
<ASSOCIATIONS>
```

ATOM

```
<ATOM pname>
```

AVALUE

```
<AVALUE asoc>
```

BACK

```
<BACK structure [count]>
```

BIND

```
<BIND [activation-atom] (bindings ...)
          [body-decl] body ...> **F
```

BIT-SYNONYM

```
<BIT-SYNONYM first synonyms ...>
```

BLOCK

```
<BLOCK (oblist ...)>
```

BOUND

```
<BOUND? atom [environment]>
```

BUZZ

```
<BUZZ atoms ...>
```

BYTE

```
<BYTE number>
```

CHECK-VERSION?

```
<CHECK-VERSION? Version-spec>
```

CHRSET

```
<CHRSET alphabet-number {string | character |
                         number | byte} ...>
```

CHTYPE

```
<CHTYPE value type-atom>
```

Change type - returns a new object that has `TYPE` `type-atom` and the same “data part” as `value`. The `PRIMTYPE` of `value` must be the same as the `TYPEPRIM` of `type-atom` otherwise an error will be generated.

There is a shortform to change type by typing `#type-atom value` instead.

Examples:

```
<CHTYPE !\A FIX>
--> 65
#FIX !\A
--> 65
#LIST [1 2 3]
--> ERROR
```

CLOSE

```
<CLOSE channel>
```

COMPILATION-FLAG

```
<COMPILATION-FLAG atom-or-string [value]>
```

COMPILATION-FLAG-DEFAULT

```
<COMPILATION-FLAG-DEFAULT atom-or-string value>
```

COMPILATION-FLAG-VALUE

```
<COMPILATION-FLAG-VALUE atom-or-string>
```

COND

```
<COND (condition body ...) ...> **F
```

CONS

```
<CONS first rest>
```

CONSTANT

```
<CONSTANT atom-or-adecl value> **F
```

CRLF

```
<CRLF [channel]>
```

DECL-CHECK

```
<DECL-CHECK boolean>
```

DECL?

```
<DECL? value pattern>
```

DEFAULT-DEFINITION

```
<DEFAULT-DEFINITION name body ...> **F
```

DEFINE

```
<DEFINE name [activation-atom] arg-list [decl] body ...> **F
```

DEFINE-GLOBALS

```
<DEFINE-GLOBALS group-name
    (atom-or-adecl [{BYTE | WORD}] [initializer]) ...> **F
```

DEFINE20

```
<DEFINE20 name [activation-atom] arg-list [decl] body ...> **F
```

DEFINITIONS

```
<DEFINITIONS package-name>
```

DEFMAC

```
<DEFMAC name [activation-atom] arg-list [decl] body ...> **F
```

DEFSTRUCT

```
<DEFSTRUCT
```

```
type-name {base-type | (base-type struct-options ...)}
(field-name decl field-options ...) ...> **F
```

DELAY-DEFINITION

```
<DELAY-DEFINITION name>
```

DIR-SYNONYM

```
<DIR-SYNONYM original synonyms ...>
```

DIRECTIONS

```
<DIRECTIONS atoms ...>
```

EMPTY?

```
<EMPTY? Structure>
```

END-DEFINITIONS

```
<END-DEFINITIONS>
```

ENDBLOCK

```
<ENDBLOCK>
```

ENDPACKAGE

```
<ENDPACKAGE>
```

ENDSECTION

```
<ENDSECTION>
```

ENTRY

```
<ENTRY atoms ...>
```

EQVB

```
<EQVB numbers ...>
```

ERROR

```
<ERROR values ...>
```

EVAL

```
<EVAL value [environment]>
```

EVALTYPE

```
<EVALTYPE atom [handler]>
```

EXPAND

```
<EXPAND value>
```

FILE-FLAGS

```
<FILE-FLAGS {CLEAN-STACK? | MDL-ZIL?} ...>
```

FILE-LENGTH

```
<FILE-LENGTH channel>
```

FLOAD

```
<FLOAD filename>
```

FORM

```
<FORM values ...>
```

FUNCTION

```
<FUNCTION [activation-atom] arg-list [decl] body ...> **F
```

FUNNY-GLOBALS?

```
<FUNNY-GLOBALS? [boolean]>
```

G=?

```
<G=? value1 value2>
```

Predicate. True if `value1` is greater or equal than `value2` otherwise false.

G?

```
<G? value1 value2>
```

Predicate. True if `value1` is greater than `value2` otherwise false.

GASSIGNED?

```
<GASSIGNED? Atom>
```

GBOUND?

```
<GBOUND? Atom>
```

GC

```
<GC>
```

GDECL

```
<GDECL (atoms ...) decl ...> **F
```

GET-DECL

```
<GET-DECL item>
```

GETB

```
<GETB table index>
```

GETPROP

```
<GETPROP item indicator [default-value]>
```

GLOBAL

```
<GLOBAL atom-or-adecl default-value [decl] [size]> **F
```

GROW

```
<GROW structure end beginning>
```

GUNASSIGN

```
<GUNASSIGN atom>
```

GVAL

```
<GVAL atom>
```

IFFLAG

```
<IFFLAG (condition body ...) ...> **F
```

ILIST

```
<ILIST count [init]>
```

IMAGE

```
<IMAGE ch [channel]>
```

INCLUDE

```
<INCLUDE package-name ...>
```

INCLUDE-WHEN

```
<INCLUDE-WHEN condition package-name ...>
```

INDENT-TO

```
<INDENT-TO position [channel]>
```

INDEX

```
<INDEX offset>
```

INDICATOR

```
<INDICATOR asoc>
```

INSERT

```
<INSERT string-or-atom oblist>
```

INSERT-FILE

```
<INSERT-FILE filename>
```

ISTRING

```
<ISTRING count [init]>
```

ITABLE

```
<ITABLE [specifier] count [(flags...)] defaults ...>
```

Defines a table of `count` elements filled with default values: either zeros or, if the `default` list is specified, the specified list of values repeated until the table is full.

The optional `specifier` may be the atoms `NONE`, `BYTE`, or `WORD`. `BYTE` and `WORD` change the type of the table and also turn on the length marker (element 0 in the table contains the length of the table), This can also be done with the flags (see `TABLE` about flags).

Examples:

```
<ITABLE 4 0>  -->
```

Element 0 WORD	Element 1 WORD	Element 2 WORD	Element 3 WORD
0	0	0	0

```
<ITABLE (BYTE LENGTH) 4 0>  -->
```

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	0	0	0	0

```
<ITABLE BYTE 4 0>  -->
```

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	0	0	0	0

ITEM

```
<ITEM asoc>
```

IVECTOR

```
<IVECTOR count [init]>
```

L=?

```
<L=? value1 value2>
```

Predicate. True if `value1` is lower or equal than `value2` otherwise false.

L?

```
<L? value1 value2>
```

Predicate. True if `value1` is lower than `value2` otherwise false.

LANGUAGE

```
<LANGUAGE name [escape-char] [change-chrset]>
```

LEGAL?

```
<LEGAL? Value>
```

LENGTH

```
<LENGTH structure>
```

LENGTH?

```
<LENGTH? structure limit>
```

LINK

```
<LINK value str oblist>
```

LIST

```
<LIST values ...>
```

LONG-WORDS?

```
<LONG-WORDS? [boolean]>
```

LOOKUP

```
<LOOKUP str oblist>
```

LPARSE

```
<LPARSE text [10] [lookup-oblist]>
```

LSH

```
<LSH number1 number2>
```

LTABLE

```
<LTABLE [flag-list] values ...>
```

Defines a table containing the specified `values` and with the `LENGTH` flag (see `TABLE`).

LVAL

```
<LVAL atom [environment]>
```

M-HPOS

```
<M-HPOS channel>
```

MAPF

```
<MAPF finalf applicable structs ...>
```

MAPLEAVE

```
<MAPLEAVE [value]>
```

MAPR

```
<MAPR finalf applicable structs ...>
```

MAPRET

```
<MAPRET [value] ...>
```

MAPSTOP

```
<MAPSTOP [value] ...>
```

MAX

```
<MAX numbers ...>
```

MEMBER

```
<MEMBER item structure>
```

MEMQ

```
<MEMQ item structure>
```

MIN

```
<MIN numbers ...>
```

MOBLIST

```
<MOBLIST name>
```

MOD

```
<MOD number1 number2>
```

MSETG

```
<MSETG atom-or-adecl value> **F
```

N==?

```
<N==? value1 value2>
```

N=?

```
<N=? value1 value2>
```

NEW-ADD-WORD

```
<NEW-ADD-WORD atom-or-string [type] [value] [flags]>
```

NEWTYPE

```
<NEWTYPE name primtype-atom [decl]>
```

NEXT

```
<NEXT asoc>
```

NOT

```
<NOT value>
```

NTH

```
<NTH structure index>
```

OBJECT

```
<OBJECT name (property values ...) ...>
```

OBLIST?

```
<OBLIST? Atom>
```

OFFSET

```
<OFFSET offset structure-decl [value-decl]>
```

OPEN

```
<OPEN "READ" path>
```

OR

```
<OR conditions ...> **F
```

OR?

```
<OR? Values ...>
```

ORB

```
<ORB numbers ...>
```

ORDER-FLAGS?

```
<ORDER-FLAGS? LAST objects ...>
```

ORDER-OBJECTS?

```
<ORDER-OBJECTS? Atom>
```

ORDER-TREE?

```
<ORDER-TREE? Atom>
```

PACKAGE

```
<PACKAGE package-name>
```

PARSE

```
<PARSE text [10] [lookup-oblist]>
```

PLTABLE

```
<PLTABLE [flags ...] values ...>
```

Defines a table containing the specified `values` and with the `LENGTH` and `PURE` flag (see `TABLE`).

PNAME

```
<PNAME atom>
```

PREP-SYNONYM

```
<PREP-SYNONYM original synonyms ...>
```

PRIMTYPE

```
<PRIMTYPE value>
```

evaluates to the primitive type of `value`. The primitive types are `ATOM`, `FIX`, `LIST`, `STRING`, `TABLE` and `VECTOR`.

Examples:

```
<PRIMTYPE !\A>
--> FIX
<PRIMTYPE <+1 2>>
--> FIX
<PRIMTYPE "ABC">
--> STRING
```

PRIN1

```
<PRIN1 value [channel]>
```

PRINC

```
<PRINC value [channel]>
```

PRINT

```
<PRINT value [channel]>
```

PRINT-MANY

```
<PRINT-MANY channel printer items ...>
```

PRINTTYPE

```
<PRINTTYPE atom [handler]>
```

PROG

```
<PROG [activation-atom] (bindings ...)
          [body-decl] body ...> **F
```

PROPDEF

```
<PROPDEF atom default-value spec ...> **F
```

PTABLE

```
<PTABLE [(flags ...)] values ...>
```

Defines a table containing the specified `values` and with the `PURE` flag (see `TABLE`).

PUT

```
<PUT structure index new-value>
```

PUT-DECL

```
<PUT-DECL item pattern>
```

PUTB

```
<PUTB table index new-value>
```

PUTPROP

```
<PUTPROP item indicator [value]>
```

PUTREST

```
<PUTREST list new-rest>
```

QUIT

```
<QUIT [exit-code]>
```

QUOTE

```
<QUOTE value> **F
```

READSTRING

```
<READSTRING dest channel [max-length-or-stop-chars]>
```

REMOVE

```
<REMOVE {atom | pname oblist}>
```

RENTRY

```
<RENTRY atoms ...>
```

REPEAT

```
<REPEAT [activation-atom] (bindings ...)
          [body-decl] body ...> **F
```

REPLACE-DEFINITION

```
<REPLACE-DEFINITION name body ...> **F
```

REST

```
<REST structure [count]>
```

RETURN

```
<RETURN [value] [activation]>
```

ROOM

```
<ROOM name (property value ...) ...>
```

ROOT

```
<ROOT>
```

ROUTINE

```
<ROUTINE name [activation-atom] arg-list body ...> **F
```

ROUTINE-FLAGS

```
<ROUTINE-FLAGS flags ...>
```

SET

```
<SET atom value [environment]>
```

SET-DEFSTRUCT-FILE-DEFAULTS

```
<SET-DEFSTRUCT-FILE-DEFAULTS args ...> **F
```

SETG

```
<SETG atom value>
```

SETG20

```
<SETG20 atom value>
```

SORT

```
<SORT predicate vector [record-size] [key-offset]
          [vector [record-size] ...]>
```

SPNAME

```
<SPNAME atom>
```

STRING

```
<STRING values ...>
```

STRUCTURED?

```
<STRUCTURED? Value>
```

SUBSTRUC

```
<SUBSTRUC structure [rest] [amount] [structure]>
```

SYNONYM

```
<SYNONYM original synonyms ...>
```

SYNTAX

```
<SYNTAX verb [prep1] [OBJECT] [(FIND flag-name)]
          [(search-flags ...)] [prep2] [OBJECT]
          [(FIND flag-name)] [(search-flags ...)]
          = action-routine-name [preaction-routine-name]
          [action-name]>
```

TABLE

```
<TABLE [(flags ...)] values ...>
```

Defines a table containing the specified `values`.

The optional `specifier` may be the atoms `NONE`, `BYTE`, or `WORD`. `BYTE` and `WORD` change the type of the table and also turn on the length marker (element 0 in the table contains the length of the table), This can also be done with the flags (see `TABLE` about flags).

These `flags` control the format of the table:

- `WORD` causes the elements to be 2-byte words. This is the default.
- `BYTE` causes the elements to be single bytes.
- `LEXV` causes the elements to be 4-byte records. If `default` values are given to `ITABLE` with this flag, they will be split into groups of three: the first compiled as a word, the next two compiled as bytes. The table is also prefixed with a byte indicating the number of records, followed by a zero byte
- `STRING` causes the elements to be single bytes and also changes the initializer format. This flag may not be used with `ITABLE`. When this flag is given, any `values` given as strings will be compiled as a series of individual ASCII characters, rather than as string addresses.

These `flags` alter the table without changing its basic format:

- `LENGTH` causes a length marker to be written at the beginning of the table, indicating the number of elements that follow. The length marker is a byte if `BYTE` or `STRING` are also given; otherwise the length marker is a `WORD`. This flag is ignored if `LEXV` is given
- `PURE` causes the table to be compiled into static memory (ROM).

The flags `LENGTH` and `PURE` are implied in `LTABLE`, `PTABLE` or `PLTABLE`.

Examples:

```
<TABLE 1 2 3 4>  -->
```

Element 0 WORD	Element 1 WORD	Element 2 WORD	Element 3 WORD
1	2	3	4

```
<TABLE (BYTE LENGTH) 1 2 3 4>  -->
```

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	1	2	3	4

```
<TELL-TOKENS {pattern form} ...> **F
<TOP structure>
```

```
<TUPLE values ...>
```

TYPE

```
<TYPE value>
```

evaluates to the type of `value`. Also see `ALLTYPES`.

Examples:

```
<TYPE !\A>
--> CHARACTER
<TYPE <+1 2>>
--> FIX
<TYPE #BYTE 42>
--> BYTE
```

TYPE?

```
<TYPE? value type-1 ... type-N>
```

Evaluates to type-i only if `<==? type-i >` is true. It is faster and gives more information than `OR`ing tests for each `TYPE`. If the test fails for all type-i's, `TYPE?` returns `#FALSE ()`.

Examples:

```
<TYPE? !\A CHARACTER FIX>
--> CHARACTER
<TYPE? <+1 2> CHARACTER FIX>
--> FIX
<TYPE? #BYTE 42 CHARACTER FIX>
--> #FALSE ()
```

TYPEPRIM

```
<TYPEPRIM type>
```

evaluates to the primitive type of `type`. The primitive types are `ATOM`, `FIX`, `LIST`, `STRING`, `TABLE` and `VECTOR`.

Examples:

```
<TYPEPRIM CHARACTER>
--> FIX
<TYPEPRIM FORM>
--> LIST
<PRIMTYPE BYTE>
--> FIX
```

UNASSIGN

```
<UNASSIGN atom [environment]>
```

UNPARSE

```
<UNPARSE value>
```

USE

```
<USE package-name ...>
```

USE-WHEN

```
<USE-WHEN condition package-name ...>
```

VALID-TYPE?

```
<VALID-TYPE? Atom>
```

VALUE

```
<VALUE atom [environment]>
```

VECTOR

```
<VECTOR values ...>
```

VERB-SYNONYM

```
<VERB-SYNONYM original synonyms ...>
```

VERSION

```
<VERSION {ZIP | EZIP | XZIP | YZIP | number} [TIME]>
```

VERSION?

```
<VERSION? (version-spec body ...) ...> **F
```

VOC

```
<VOC string [part-of-speech]>
```

XORB

```
<XORB numbers ...>
```

ZGET

```
<ZGET table index>
```

ZIP-OPTIONS

```
<ZIP-OPTIONS {COLOR | MOUSE | UNDO | DISPLAY | SOUND
              | MENU} ...>
```

ZPUT

```
<ZPUT table index new-value>
```

ZREST

```
<ZREST table bytes>
```

ZSTART

```
<ZSTART atom>
```

=== *Z-code builtins* ===

(i.e. "things you can use inside a routine")

Sources:

The Z-Machine Standards Document, Graham Nelson

The Inform Designer's Manual, Graham Nelson

ZIL Language Guide, Jesse McGrew

*, MUL

```
<* numbers ...>
```

Zapf syntax	**Inform syntax**
MUL	mul

Multiply `numbers`.

Example:

```
<* 2 3 4> -->  24
```

+, ADD

```
<+ numbers ...>
```

Zapf syntax	**Inform syntax**
ADD	add

All versions

Add `numbers`.

Example:

```
<+ 2 3 4> -->  7
```

-, SUB

```
<- numbers ...>
```

Zapf syntax	**Inform syntax**
SUB	sub

All versions

Subtract first `number` by subsequent `numbers`.

Example:

```
<* 8 3 4> -->  1
```

/, DIV

```
</ numbers ...>
```

Zapf syntax	**Inform syntax**

```
DIV                 div

All versions
```

Divide first `number` by subsequent `numbers`.

Example:

```
<* 20 5 2>      -->  2
```

0?, ZERO?

```
<0? value>
```

Zapf syntax	**Inform syntax**
ZERO?	Jz

```
All versions
```

Predicate. True if `value` is 0 otherwise false.

Example:

```
<0? <- 1 1>>    -->  TRUE
```

1?

```
<1? value>
```

Predicate. True if `value` is 1 otherwise false.

=?, ==?, EQUAL?

```
<=? value1 value2...valueN>
```

Zapf syntax	**Inform syntax**
EQUAL?	Je

```
All versions
```

Predicate. True if `value1` is equal to any of the values `value2` to `valueN`.

Examples:

```
<=? 1 1>        -->  TRUE
<=? 1 2>        -->  FALSE
<=? 1 2 1>      -->  TRUE
```

AGAIN

```
<AGAIN [activation]>
```

Skips the rest of the loop and starts again from the top.

AND

```
<AND expressions...>
```

Logical AND.

APPLY

```
<APPLY routine values...>
```

ASH, ASHIFT

```
<ASH number places>
```

Zapf syntax	Inform syntax
ASHIFT	art_shift

Versions: 5-

Arithmetic shift. Shifts `number` left when `places` is positive and right if it is negative. When right shift the sign is preserved (if bit 15 is 1 a 1 is shifted in, otherwise a 0 is shifted in).

```
1000 0000 0000 1010     -->  1100 0000 0000 0101
```

Also see `LSH`.

Examples:

```
<ASH 4 1>      -->  8
<ASH 4 -2>     -->  1
```

ASSIGNED?

```
<ASSIGNED? Name>
```

Zapf syntax	Inform syntax
ASSIGNED?	check_arg_count

Versions: 5-

Predicate. Can test if optional argument `name` in call to routine is supplied.

Example:

```
<ROUTINE TEST("OPT" X)
     <COND (<ASSIGNED? X>
          <TELL "X is assigned." CR>
     )
     (ELSE
          <TELL "X is not assigned." CR>
     )>
>

<TEST>         --> X is not assigned.
<TEST 1>       --> X is assigned.
```

BACK

```
<BACK table [bytes]>
```

BAND, ANDB

```
<BAND numbers...>
```

Zapf syntax	**Inform syntax**
BAND	and

All versions

Bitwise AND.

Examples:

```
<BAND 33 96>        -->  32
<BAND 33 96 64>     -->  0
```

BCOM

```
<BCOM value>
```

Zapf syntax	**Inform syntax**
BCOM	not

All versions

Bitwise NOT. Reverse all bits in the WORD `value` (16 bits).

Examples:

```
<BCOM #2 000011110001111>     --> #2 1111000011110000
```

BIND

```
<BIND (bindings...) expressions...>
```

BOR, ORB

```
<BOR numbers...>
```

Zapf syntax	**Inform syntax**
BOR	or

All versions

Bitwise OR.

Examples:

```
<BOR 33 96>    -->  97
<BOR 33 96 64> -->  97
```

BTST

```
<BTST value1 value2>
```

Zapf syntax	**Inform syntax**
BTST	test

```
All versions
```

Predicate. Binary test. Evaluates to true if all `value2` bits are set in `value1`. Could be expressed as `<=? <BAND value1 value2> value2>`.

Examples:

```
<BTST 64 64>   --> TRUE
<BTST 64 63>   --> FALSE
<BTST 97 33>   --> TRUE
```

BUFOUT

```
<BUFOUT value>
```

Zapf syntax	**Inform syntax**
BUFOUT	buffer_mode

```
Versions: 4-
```

Flag that controls if output is buffered (to enable proper word-wrap). `Value` can be true or false.

Examples:

```
<BUFOUT <>>    --> Turns off buffering(disables word-wrap)
<BUFOUT T>     --> Turns on buffering
```

CATCH

```
<CATCH>
```

Zapf syntax	**Inform syntax**
CATCH	catch

```
Versions: 5-
```

Used in conjunction with `THROW`. `CATCH` returns the current state of the stack (the "stack frame"). Also see `THROW`.

Example:

```
<SETG CATCH-POINT <CATCH>>    -->  Saves the current stack
                                   frame in global variable
```

CHECKU

```
<CHECKU character>
```

Zapf syntax	**Inform syntax**
CHECKU	check_unicode

```
Versions: 5-
```

Checks if given unicode `character` can be printed and/or receieved from keyboard. Return is in

bit 0 and 1 so the return result is either 0, 1, 2 or 3.

0 = character can not be printed and not recieved from keyboard
1 = character can be printed but not recieved from keyboard
2 = character can not be printed but recieved from keyboard
3 = character can both be printed and recieved from keyboard

Example:

```
<CHECKU 65>    -->  3
```

CLEAR

```
<CLEAR window-number>
```

Zapf syntax	**Inform syntax**
CLEAR	erase_window

Versions: 4-

Clears window with given `window-number`. If `window-number` is -1 it unsplit all windows and then clears the resulting window. If `window-number` is -2 it clears all windows without unsplitting.

Example:

```
<CLEAR 0>      --> Clears window 0 (the "main"-window)
```

COLOR

```
<COLOR fg-color bg-color>                    ;"Version 5"
<COLOR fg-color bg-color [window-number]>    ;"Versions: 6-"
```

Zapf syntax	**Inform syntax**
COLOR	set_colour

Versions: 5-

Print text in given `fg-color` and `bg-color` from this point on (flushing out text in buffer in old colors first). Version 6 supports a third argument, `window-number`. The colors available (if interpreter supports it) are:

0	Current color
1	Default color
2	Black
3	Red
4	Green
5	Yellow
6	Blue
7	Magenta
8	Cyan
9	White

Example:

```
<COLOR 2 9>    -->  Set black text against white background
```

COND

```
<COND (condition expressions...)...>
```

COPYT

```
<COPYT src-table dest-table length>
```

Zapf syntax	**Inform syntax**
COPYT	copy_table

Versions: 5-

Copies `length` number of bytes from `src-table` to `dest-table`. The tables are allowed to overlap. If `length` is positive then the copy is done without corrupting the `src-table`. If `length` is negative the copy is always forward from `src-table` to `dest-table` (the absolute `length` number of bytes) even if this corrupts `src-table`.

Example:

```
<GLOBAL TABLE1 <TABLE 1 2 3>>
<GLOBAL TABLE2 <TABLE 0 0 0>>
<ROUTINE TEST-COPYT()
     <COPYT ,TABLE1 ,TABLE2 6>
     <GET ,TABLE2 2>
>

<TEST-COPYT>   -->  3
```

CRLF

```
<CRLF>
```

Zapf syntax	**Inform syntax**
CRLF	new_line

All versions

Prints carriage return and line feed.

Example:

```
<CRLF>    -->  Moves curser to position 1 on new line
```

CURGET

```
<CURGET table>
```

Zapf syntax	**Inform syntax**
CURGET	get_cursor

```
Versions: 4-
```

CURGET puts current cursor row in record 0 and current cursor column in record 1 of supplied table.Both row and column are WORD (16-bit).

Example:

```
<GLOBAL CURTABLE <TABLE 0 0>>
<ROUTINE TEST-CURGET ()
     <CURGET ,CURTABLE>
>

<TEST-CURGET>  -->  Puts current row and column in CURTABLE
```

CURSET

```
<CURSET row column>                      ;"Versions: 4-5"
<CURSET row column [window-number]>      ;"Versions: 6-"

Versions: 4-
```

CURSET moves cursor to row and column in current window (or supplied window-number). In versions 6-, if row is -1 then the cursor is turned off (-2 turns it back on).

Example:

```
<CURSET 1 1>   -->  Move cursor to upper left corner in
                    current window
```

DCLEAR

```
<DCLEAR picture-number [row] [column]>

Zapf syntax          Inform syntax
DCLEAR               erase_picture

Versions: 6-
```

Clears (draw background color) area covered by picture-number, starting at row and column. Also see DISPLAY.

Example:

```
<DCLEAR 1 1 1>      --> Clears picture 1
```

DEC

```
<DEC name>

Zapf syntax          Inform syntax
DEC                  dec

All versions
```

Decrease variable (signed) name with 1.

Example:

```
<ROUTINE TEST-DEC (X) <DEC .X>>

<TEST-DEC 45>       -->  44
<TEST-DEC 0>        -->  -1
```

DIRIN

```
<DIRIN stream-number>
```

Zapf syntax	**Inform syntax**
DIRIN	input_stream

All versions

Select input stream. Only `stream-number` 0 and 1 are valid.

0	Keyboard
1	File on host

Example:

```
<DIRIN 0>      -->  True and select input stream keyboard
```

DIROUT

```
<DIROUT stream-number [table]>          ;"Versions -5"
<DIROUT stream-number [table] [width]> ;"Versions 6-"
```

Zapf syntax	**Inform syntax**
DIROUT	output_stream

Directs output to one or more output streams (multiple streams can be active simultaneously). Turn on stream with positive `stream-number` and turn off stream with negative `stream-number`.

If stream 3 is active a `table` must be supplied. WORD 0 in `table` holds number of printed characters and byte 2 onward holds the characters printed. `DIROUT` can overrun table if not enough space is allocated.

Later versions can format output text to `width` (number of characters if `width` is positive or number of pixels if `width` is negative).

1	Screen
2	File on host (transcript)
3	Table
4	File of commands on host

Example:

```
<DIROUT 3>     -->  Turns on output to file
<DIROUT -3>    -->  Turns off output to file
```

DISPLAY

```
<DISPLAY picture-number [row] [column]>
```

Zapf syntax	**Inform syntax**
DISPLAY	draw_picture

Versions: 6-

Draw picture-number at coordinates row and column. If row and column are omitted current cursor positionis used.

Example:

```
<DISPLAY 1>    --> Draws picture 1 at current cursor position
```

DLESS?

```
<DLESS? name value>
```

Zapf syntax	**Inform syntax**
DLESS?	dec_chk

All versions

Predicate. Decrease variable (signed) `name` with 1and returns true if variable `name` is lower than `value`, otherwise returns false.

Example:

```
<ROUTINE TEST-DLESS? (X)
     <PRINTN <DLESS? X 100>>
     <CRLF>
     <PRINTN .X>
>

<TEST-DLESS? 101>   -->  "0\n100"
```

DO

```
<DO (name start end [step]) expressions...>
```

A quirk of the DO statement, which can be thought of as a cross between a Pascal-style "for" statement and a C-style "for" statement.

Pascal-style "for" statements loop over a range of values:

```
// Pascal
for i := 1 to 10 do ...
for j := 10 downto 1 do ...
```

```
// ZIL
<DO (I 1 10) ...>
<DO (J 10 1 -1) ...>
```

C-style "for" statements initialize some state, then mutate it and repeat until a condition becomes false. In ZIL, the condition is reversed - the loop exits when it becomes true:

```
// C
for (i = first(obj); i; i = next(i)) { ... }
```

```
// ZIL
<DO (I <FIRST? .OBJ> <NOT .I> <NEXT? .I>) ...>
```

Notice that every Pascal-style loop can be transformed into a C-style loop:

```
// Pascal-style loops
<DO (I 1 10) ...>
<DO (J 10 1 -1) ...>

// C-style equivalents
<DO (I 1 <G? .I 10> <+ .I 1>) ...>
<DO (J 10 <L? .J 1> <- .J 1>) ...>
```

The quirk is that the behavior of DO depends on the syntax you use for each part.

If the third value inside the parens is a complex FORM -- meaning one that isn't a simple LVAL or GVAL, like '.MAX' is -- it's assumed to be a "C-style" exit condition, otherwise it's assumed to be a "Pascal-style" upper/lower bound. Likewise, the optional fourth value is treated as either a C-style mutator or a Pascal-style step size.

More of the DO statement's quirks are demonstrated here:

```
<ROUTINE GO ()
    <TEST-PASCAL-STYLE>
    <TEST-C-STYLE>
    <TEST-MIXED-STYLE>
    <QUIT>>

<CONSTANT C-ONE 1>
<CONSTANT C-TEN 10>

<ROUTINE TEST-PASCAL-STYLE ("AUX" (ONE 1) (TEN 10))
    <TELL "== Pascal style ==" CR>

    <TELL "Counting from 1 to 10...">
    ;"1 2 3 4 5 6 7 8 9 10"
    <DO (I 1 10)
        (END <CRLF>)
        <TELL " " N .I>>

    <TELL "Counting from 1 to 10 with step 2...">
    ;"1 3 5 7 9"
    <DO (I 1 10 2)
        (END <CRLF>)
        <TELL " " N .I>>

    <TELL "Counting from 10 to 1...">
    ;"10 9 8 7 6 5 4 3 2 1"
    <DO (I 10 1)
        (END <CRLF>)
        <TELL " " N .I>>

    <TELL "Counting from 10 to 1 with step -2...">
    ;"10 8 6 4 2"
    <DO (I 10 1 -2)
        (END <CRLF>)
        <TELL " " N .I>>
```

```
<TELL "Counting from .ONE to .TEN...">
;"1 2 3 4 5 6 7 8 9 10"
<DO (I .ONE .TEN)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from .TEN to .ONE...">
;"10"
;"Since the loop bounds aren't FIXes (numeric
 literals), ZILF doesn't know the loop is meant
 to count down, and it compiles a loop that counts
 up and exits after the first iteration. A DO loop
 whose condition is a constant or simple FORM always
 runs at least once."
<DO (I .TEN .ONE)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from 10 to .ONE...">
;"10"
;"See above."
<DO (I 10 .ONE)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from .TEN to 1...">
;"10"
;"See above."
<DO (I .TEN 1)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from .TEN to .ONE with step -1...">
;"10 9 8 7 6 5 4 3 2 1"
<DO (I .TEN .ONE -1)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from ,C-TEN to ,C-ONE...">
;"10"
;"Even defining the loop bounds as CONSTANTs won't
 tell ZILF that the loop needs to run backwards."
<DO (I ,C-TEN ,C-ONE)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from %,C-TEN to %,C-ONE...">
;"10 9 8 7 5 4 3 2 1"
;"The % forces ,C-TEN to be evaluated at read time,
 so the loop bounds are specified as FIXes, allowing
 ZILF to determine that the loop runs backwards."
<DO (I %,C-TEN %,C-ONE)
```

```
        (END <CRLF>)
        <TELL " " N .I>>

    <CRLF>>

<OBJECT DESK
    (DESC "desk")>

<OBJECT MONITOR
    (DESC "monitor")
    (LOC DESK)>

<OBJECT KEYBOARD
    (DESC "keyboard")
    (LOC DESK)>

<OBJECT MOUSE
    (DESC "mouse")
    (LOC DESK)>

<ROUTINE TEST-C-STYLE ()
    <TELL "== C style ==" CR>

    <TELL "Counting from 10 down to 1...">
    ;"10 9 8 7 6 5 4 3 2 1"
    <DO (I 10 <L? .I 1> <- .I 1>)
        (END <CRLF>)
        <TELL " " N .I>>

    <TELL "Counting from 10 up (!) to 1...">
    ;""
    ;"Nothing is printed, because the exit condition
     is initially true. A DO loop whose condition is
     a complex FORM can exit before the first iteration."
    <DO (I 10 <G? .I 1> <+ .I 1>)
        (END <CRLF>)
        <TELL " " N .I>>

    <TELL "On the desk:">
    ;"monitor mouse keyboard"
    <DO (I <FIRST? ,DESK> <NOT .I> <NEXT? .I>)
        (END <CRLF>)
        <TELL " " D .I>>

    <CRLF>>


<ROUTINE TEST-MIXED-STYLE ()
    <TELL "== Mixed ==" CR>

    <TELL "Powers of 2 up to 1000:">
    ;"1 2 4 8 16 32 64 128 256 512"
    <DO (I 1 1000 <* .I 2>)
        (END <CRLF>)
```

```
            <TELL " " N .I>>

        <CRLF>>
```

Highlights:

- Loops can include subsequent code in an (END ...) clause for brevity, e.g. to print a newline after a list.

A Pascal-style DO can *sometimes* determine when it needs to run backwards, even if no step size is provided.

Pascal and C style can be mixed in the same loop, e.g. <DO (I 1 1000 <* .I 2>) ...> to count powers of 2 up to 1000.

ERASE

```
<ERASE value>
```

Zapf syntax	**Inform syntax**
ERASE	erase_line

```
Versions: 4-
```

Versions 4 and 5: if the `value` is 1, erase from the current cursor position to the end of its line in the current window. If the `value` is anything other than 1, do nothing.

Version 6: if the `value` is 1, erase from the current cursor position to the end of the its line in the current window. If not, erase the given number of pixels minus one across from the cursor (clipped to stay inside the right margin). The cursor does not move.

Example:

```
<ERASE 1>      -->  Clears from cursor to end of line
```

F?

```
<F? expression>
```

FCLEAR

```
<FCLEAR object flag>
```

Zapf syntax	**Inform syntax**
FCLEAR	clear_attr

```
All versions
```

Removes `flag` from `object`.

Example:

```
<FCLEAR ,TRAP-DOOR ,OPENBIT>  -->  Marks the trap-door as
                                   closed
```

FIRST?

```
<FIRST? object>
```

Zapf syntax	Inform syntax
FIRST?	get_child

All versions

Returns first object inside (contained) in `object`. Returns 0 (false) if no object exists.

Example:

```
<SET RM <FIRST? ,ROOMS>> -->  Sets RM to first object in
                              ROOMS. Also evaluates to
                              true (all values not 0 is true)
```

FONT

```
<FONT number>                        ;"Version 5"
<FONT number [window-number]>        ;"Versions 6-"
```

Zapf syntax	Inform syntax
FONT	set_font

Versions: 5-

Sets current font to `number`. Returns old fonts `number`. If font `number` is not available 0 (false) is returned.

1	Normal font
3	Character graphics font (see §16 in *The Z-Machine Standards Document*)
4	Monospace (fixed-pitch) font

Example:

```
<FONT 4>  -->  Sets fixed-pitch font. In version 3-4 this is
               done by setting bit 1 of Flags 2 in header
               <PUT 0 8 <BOR <GET 0 8> 2>>
```

FSET

```
<FSET object flag>
```

Zapf syntax	Inform syntax
FSET	set_attr

All versions

Add `flag` to `object`.

Example:

```
<FSET ,TRAP-DOOR ,OPENBIT>    -->  Marks the trap-door as
                                   open
```

FSET?

```
<FSET? object flag>
```

Zapf syntax	Inform syntax
FSET?	test_attr

All versions

Predicate. Tests if `flag` set on `object`.

Example:

```
<FSET? ,TRAP-DOOR ,OPENBIT>  -->  True if OPENBIT is set
```

FSTACK

```
<FSTACK number [stack]>
```

Zapf syntax	Inform syntax
FSTACK	pop / pop_stack

Versions: 6-

Removes `number` of items from system stack or given `stack` (table).

Example:

```
<PUSH 123> <PUSH 0> <PUSH 0> <PUSH 0> <FSTACK 3> <POP>
     ---> 123
```

G?, GRTR?

```
<G? value1 value2>
```

Zapf syntax	Inform syntax
GRTR?	Jg

All versions

Predicate. Returns true if `value1` is greater than `value2`, otherwise false.

Examples:

```
<G? 5 4>  -->  T
<G? 4 5>  -->  <>
```

G=?

```
<G=? value value>
```

GET

```
<GET table offset>
```

Zapf syntax	Inform syntax
GET	loadw

```
All versions
```

Returns WORD-record (2 bytes) stored at offset.
Note: table is an address in memory so the WORD that is returned is at table+offset*2. It is legal to use, for example, 0 as address to retrieve information from header.

Example:

```
<GET <TABLE 0 1 2 3> 2>       -->  2
```

GETB

```
<GETB table offset>
```

Zapf syntax	**Inform syntax**
GETB	loadb

```
All versions
```

Returns BYTE-record (1 byte) stored at offset.
Note: table is an address in memory so the BYTE that is returned is at table+offset. It is legal to use, for example, 0 as address to retrieve information from header.

Example:

```
<GETB <TABLE (BYTE) !\A !\B !\C !\D> 2>     -->  !\C
```

GETP

```
<GETP object property>
```

Zapf syntax	**Inform syntax**
GETP	get_prop

```
All versions
```

Get property from object. Returns default value if property is not declared in object.

Example:

```
<OBJECT MYOBJ (MYPROP 123)>

<GETP ,MYOBJ ,P?MYPROP>  -->  123
```

GETPT

```
<GETPT object property>
```

Zapf syntax	**Inform syntax**
GETPT	get_prop_addr

```
All versions
```

Get property adress from object. Returns 0 (false) if property is not declared in object.

Example:

```
<OBJECT MYOBJ (MYPROP 123)>

<GET <GETPT ,MYOBJ ,P?MYPROP> 0>   -->  123
<GETPT ,MYOBJ ,P?MYPROP2>          -->  0
```

GVAL

```
<GVAL name>
```

HLIGHT

```
<HLIGHT style>
```

Zapf syntax	**Inform syntax**
HLIGHT	set_text_style

Versions: 4-

Set text to style. It ispossible to combine styles.

0	Normal
1	Inverse
2	Bold
4	Italic
8	Monospace

Example:

```
<HLIGHT 2>          -->  Set font to bold
```

IFFLAG

```
<IFFLAG (compilation-flag-condition expressions...)...>
```

IGRTR?

```
<IGRTR? name value>
```

Zapf syntax	**Inform syntax**
IGRTR?	inc_chk

All versions

Predicate. Increase variable (signed) name with 1and returns true if variable name is lower than value, otherwise returns false.

Example:

```
<ROUTINE TEST-IGRTR? (X)
     <PRINTN <IGRTR? X 100>>
     <CRLF>
     <PRINTN .X>
>
```

```
<TEST-IGRTR? 100>   -->  "1\n101"
<TEST-IGRTR? 99>    -->  "0\n100"
```

IN?

```
<IN? object1 object2>
```

Zapf syntax	Inform syntax
IN?	jin

All versions

Predicate. Returns true if `object1` is in `object2` (`object1` has `object2` as parent),otherwise false.

Example:

```
<OBJECT ANIMAL>
<OBJECT CAT (LOC ANIMAL)>

<IN? ,CAT ,ANIMAL>    -->    T
<IN? ,ANIMAL ,CAT>    -->    <>
```

INC

```
<INC name>
```

Zapf syntax	Inform syntax
INC	inc

All versions

Increment `name` by 1. (This is signed, so -1 increments to 0)

Example:

```
<GLOBAL X 5>

<INC ,X>  -->  X=6
```

INPUT

```
<INPUT 1 [time] [routine]>
```

Zapf syntax	Inform syntax
INPUT	read_char

Versions: 4-

`INPUT` reads a single character from the keyboard. Calls routine every time*0.1 s. If routine returns true input is aborted.

Examples:

```
<INPUT 1> -->  Wait for keypress
```

```
<ROUTINE WAIT-TWO-SECONDS ()
     <INPUT 1 20 ABORT-WAIT>
>

<ROUTINE ABORT-WAIT () <RETURN T>>

<WAIT-TWO-SECONDS>  -->  Pause two seconds (if not
                         interrupted by a keypress
                         from the keyboard
```

INTBL?

```
<INTBL? value table length [form]>     ;"Version 5"
<INTBL? value table length>            ;"Version 4, 6-"
```

Zapf syntax	**Inform syntax**
INTBL?	scan_table

Versions: 4-

Predicate. Returns value if value is in table of length, otherwise 0.

In version 5 form describes the field where bit 7 is set for words and clear for bytes, rest defines the length of field.

Examples:

```
<INTBLE? 3 <TABLE 1 2 3 4> 4> -->  3
<INTBLE? 6 <TABLE 1 2 3 4> 4> -->  0
<INTBL? 8 <TABLE (BYTE) 2 0 1 4 0 1 8 0 1> 9 3>  -->  8
                                                      ;"Ver 5"
```

IRESTORE

```
<IRESTORE>
```

Zapf syntax	**Inform syntax**
IRESTORE	restore_undo

Versions: 5-

Restores game state saved to memory by `ISAVE` (undo).

ISAVE

```
<ISAVE>
```

Zapf syntax	**Inform syntax**
ISAVE	save_undo

Versions: 5-

Save game state to memory that later can be restored by `IRESTORE` (undo).

ITABLE

```
<ITABLE [length-spec] number [(table-flags...)]
          [const-expressions...]>
```

L?, LESS?

```
<L? value1 value2>
```

Zapf syntax	**Inform syntax**
LESS?	Jl

All versions

Predicate. Returns true if `value1` is less than `value2`, otherwise false.

Examples:

```
<L? 5 4>  -->  <>
<L? 4 5>  -->  T
```

L=?

```
<L=? value1 value2>
```

Is `value1` lower or equal to `value2`.

LEX

```
<LEX text parse [dictionary] [flag]>
```

Zapf syntax	**Inform syntax**
LEX	tokenise

Versions: 4-

Parse the `text` into `parse`. See `READ` for more info about parsing. The game dictionary is used if not a `dictionary` table (`LTABLE`) is supplied. If the length of the `dictionary` is negative, the `dictionary` can be unsorted. If `flag` is set (true), unrecognized words are not written to `parse` but their slot is left unmodified. This makes it possible to run `LEX` against different dictionaries serially. Also see `READ`.

Example:

```
<GLOBAL TEXTBUF <TABLE (BYTE) !\c !\a !\t>>
<GLOBAL PARSEBUF <ITABLE 1 (LEXV) 0 0>>
<OBJECT CAT (SYNONYM CAT)>

<LEX ,TEXTBUF ,PARSEBUF>
<PRINTB <GET ,PARSEBUF 1>>    -->  "cat"
```

LOC

```
<LOC object>
```

Zapf syntax	**Inform syntax**

```
LOC                 get_parent

All versions
```

Returns parent to `object`.

Examples:

```
<OBJECT ANIMAL>
<OBJECT CAT (LOC ANIMAL)>

<=? <LOC ,CAT> ,ANIMAL>  -->  T
<LOC ,ANIMAL>            -->  0
```

LOWCORE-TABLE

```
<LOWCORE-TABLE field-spec length routine>
```

LOWCORE

```
<LOWCORE field-spec [new-value]>
```

LSH, SHIFT

```
<LSH number places>
```

Zapf syntax	**Inform syntax**
SHIFT	log_shift

```
Versions: 5-
```

Logical shift. Shifts `number` left when `places` is positive and right if it is negative. When right shift the sign is not preserved (0 is always shifted in).

```
1000 0000 0000 1010     -->  0100 0000 0000 0101
```

Also see A`SH`.

Examples:

```
<ASH 4 1>      -->  8
<ASH 4 -2>     -->  1
```

LTABLE

```
<LTABLE [(table-flags...)] values...>
```

LVAL

```
<LVAL name>
```

MAP-CONTENTS

```
<MAP-CONTENTS (name [next] object) expressions...>
```

MAP-DIRECTIONS

```
<MAP-DIRECTIONS (name pt room) expressions...>
```

MARGIN

```
<MARGIN left right [window-number]>
```

Zapf syntax	**Inform syntax**
MARGIN	set_margins

Versions: 6-

Set `left` and `right` margin (in pixels) in given `window-number`. If no `window-number` is specified `MARGIN` sets margins in `window-number` 0.

Example:

```
<MARGIN 1 1>   --> set 1 pixel margin in window 0
```

MENU

```
<MENU number table>
```

Zapf syntax	**Inform syntax**
MENU	make_menu

Versions: 6-

Controls menu 3- (not menu 0-2, they are system menus). The `table` is a `LTABLE` of `LTABLE`. Item 1 being the menu name. Item 2- are the entries.

Example (from Journey):

```
<GLOBAL MAC-SPECIAL-MENU
     <LTABLE <TABLE (STRING LENGTH) "Journey">
          <TABLE (STRING LENGTH) "Essences">
          <TABLE (STRING LENGTH) "No Defaults">>>
...
<MENU 3 ,MAC-SPECIAL-MENU>
```

MOD

```
<MOD number1 number2>
```

Zapf syntax	**Inform syntax**
MOD	mod

All versions

Returns remainder of 16-bit signed division. `number2` is not allowed to be 0 ("Division by zero").

Examples:

```
<MOD 15 4>     -->  3
<MOD -15 4>    -->  -3
<MOD -15 -4>   -->  -3
<MOD 15 -4>    -->  3
```

MOUSE-INFO

```
<MOUSE-INFO table>

Zapf syntax          Inform syntax
MOUSE-INFO           read_mouse

Versions: 6-
```

Reads mouse information into `table`. The table is 4 WORDS (2 bytes) long.

0	Y coordinate
1	X coordinate
2	Button bits (host dependent)
3	Menu (number*256+entry)

Example (from Journey):

```
<GLOBAL MOUSE-INFO-TBL <TABLE 0 0 0 0>>
...
<MOUSE-INFO ,MOUSE-INFO-TBL>
```

MOUSE-LIMIT

```
<MOUSE-LIMIT window-number>

Zapf syntax          Inform syntax
MOUSE-LIMIT          mouse_window

Versions: 6-
```

Restricts mouse movement to `window-number`. If `window-number` is -1 all restrictions are removed. 1 is default `window-number`.

Example:

```
<MOUSE-LIMIT 1>          -->  Mouse constrained to window 1
```

MOVE

```
<MOVE object1 object2>

Zapf syntax          Inform syntax
MOVE                 insert_obj

All versions
```

Move `object1` to be first child of `object2`. Children of `object1` moves with it.

Example:

```
<OBJECT ANIMAL>
<OBJECT CAT>
```

```
<MOVE ,CAT ,ANIMAL>
<IN? ,CAT ,ANIMAL>  -->  T
```

N=?, N==?

```
<N=? value values...>
```

NEXT?

```
<NEXT? object>
```

Zapf syntax	**Inform syntax**
NEXT?	get_sibling

All versions

Returns object after `object` in object-list (sibling). Returns 0 (false) if no object exists.

Example:

```
<OBJECT ANIMAL>
<OBJECT CAT>
<OBJECT DOG>

<MOVE ,CAT ,ANIMAL>
<MOVE ,DOG ,ANIMAL>
<=? <NEXT? ,DOG> ,CAT>        -->  T
```

NEXTP

```
<NEXTP object property>
```

Zapf syntax	**Inform syntax**
NEXTP	get_next_prop

All versions

Returns the property that comes after `property` on `object`. Returns 0 if there is no more properties after `property`. If `property` is 0 then `NEXTP` returns first property on `object`.

Example:

```
<OBJECT MYOBJ (FOO 123) (BAR 456)>

<=? <NEXTP 0> ,FOO>      -->  T
<=? <NEXTP ,FOO> ,BAR>   -->  T
<NEXTP ,BAR>             -->  0 (false)
```

NOT

```
<NOT expression>
```

Logical NOT.

OR

```
<OR expressions...>
```

Logical AND.

ORIGINAL?

```
<ORIGINAL?>

Zapf syntax          Inform syntax
ORIGINAL?            Piracy

Versions: 5-
```

Predicate. Tests if game disc is an original. Almost all modern interpreters always return true.

PICINF

```
<PICINF picture-number table>

Zapf syntax          Inform syntax
PICINF               picture_data

Versions: 6-
```

Writes picture data from `picture-number` into `table`. Word 0 of `table` holds picture width and word 1 holds picture height. Then follows the picture data.

If `picture-number` is 0, the number of available pictures is written into word 0 of `table` and release number of picture file is written into word 1.

Example:

```
<GLOBAL MYPIC <ITABLE 2048 0>>

<PICINFO 1 ,MYPIC>  -->  Writes picture data into MYPIC
```

PICSET

```
<PICSET table>

Zapf syntax          Inform syntax
PICSET               picture_table

Versions: 6-
```

Give interpreter a `table` of picture numbers that the interpreter can then unpack from disc and cache in memory.

PLTABLE

```
<PLTABLE [(table-flags...)] values...>
```

POP

```
<POP [stack]>
```

Zapf syntax	**Inform syntax**
POP	pull

Versions: 6-

Pops value of stack. If no stack is given value is popped from game stack.

Example:

```
<PUSH 123>
<POP>                --> 123

<GLOBAL MY-STACK <TABLE 3 0 0 123>>
<POP ,MY-STACK>      --> 123
```

PRINT

```
<PRINT packed-string>
```

Zapf syntax	**Inform syntax**
PRINT	print_paddr

All versions

Print `packed-string` from high memory (packed adress).

Example:

```
<GLOBAL MSG "Hello, sailor!">
<PRINT ,MSG>         --> "Hello, sailor!"
```

PRINTB

```
<PRINTB unpacked-string>
```

Zapf syntax	**Inform syntax**
PRINTB	print_addr

All versions

Print unpacked-string from dynamic or static memory (unpacked adress).

Example:

```
<OBJECT MYOBJECT (SYNONYM HELLO)>

<PRINTB <GETP ,MYOBJECT ,P?SYNONYM>>   --> "hello"
```

PRINTC

```
<PRINTC character>
```

Zapf syntax	**Inform syntax**

```
PRINTC              print_char

All versions
```

Print `character`.

Example:

```
<PRINTC 65>    -->  A
```

PRINTD

```
<PRINTD object>

Zapf syntax         Inform syntax
PRINTD              print_obj

All versions
```

Print description of `object`.

Example:

```
<GLOBAL MYOBJECT (DESC "sword">

<PRINTD ,MYOBJECT>  -->  "sword"
```

PRINTF

```
<PRINTF table>

Zapf syntax         Inform syntax
PRINTF              print_form

Versions: 6-
```

Print a formatted `table`. Each line starts with a WORD that is the number of characers that follows. Last byte in each line is 0.

PRINTI

```
<PRINTI string>

Zapf syntax         Inform syntax
PRINTI              print

All versions
```

Print `string`.

Example:

```
<PRINTI "Hello, sailor!">     -->  "Hello, sailor!"
```

PRINTN

```
<PRINTN number>
```

```
     Zapf syntax          Inform syntax
     PRINTN               print_num

     All versions

Print number.
Example:
     <PRINTN <+ 1 3>>     -->  4
     <PRINTN -42>         -->  -42
```

PRINTR

```
     <PRINTR string>

     Zapf syntax          Inform syntax
     PRINTR               print_ret

     All versions
```

Print `string` and then `CRLF`.

Example:

```
     <PRINTR "Hello. Sailor!">     -->  "Hello, sailor!\n"
```

PRINTT

```
     <PRINTT table width [height] [skip]>

     Zapf syntax          Inform syntax
     PRINTT               print_table

     Versions: 5-
```

Print `table` (string) in rectangle defined by `width` and `height`. Default `height` is 1. If `skip` is given then that number of characters is skipped between lines.

Examples:

```
     <GLOBAL MYTEXT <TABLE (STRING) "hansprestige">>

     <PRINTT ,MYTEXT 6>       -->  "hanspr\n"
     <PRINTT ,MYTEXT 4 3>     -->  "hans\npres\ntige\n"
     <PRINTT ,MYTEXT 3 3 1>   -->  "han\npre\ntig\n"
```

PRINTU

```
     <PRINTU number>

     Zapf syntax          Inform syntax
     PRINTU               print_unicode

     Versions: 5-
```

Print unicode-character `number`.

Examples:

```
<PRINTU 65>          -->   A
<PRINTU 196>         -->   Ä
```

PROG

```
<PROG (bindings...) expressions...>
```

PTABLE

```
<PTABLE [(table-flags...)] values...>
```

PTSIZE

```
<PTSIZE property-address>
```

Zapf syntax	**Inform syntax**
PTSIZE	get_prop_len

All versions

Get size in bytes of property at `property-address`.

Example:

```
<OBJECT MYOBJECT (FOO 1 2 3)>

<PTSIZE <GETPT ,MYOBJECT ,P?FOO>>      -->  6
```

PUSH

```
<PUSH value>
```

Zapf syntax	**Inform syntax**
PUSH	push

All versions

Push `value` on game stack.

Example:

```
<PUSH 123>
```

PUT

```
<PUT table offset value>
```

Zapf syntax	**Inform syntax**
PUT	storew

All versions

Put a 16-bit WORD `value` in `table` at word position `offset`. Actual address is table-address+offset*2.

Note that `table` can be a byte-address in dynamic memory.

Examples:

```
<PUT ,MYTABLE 1 123>     -->  Stores 123 at position 1
                              in MYTABLE
<PUT 0 8 <BOR <GET 0 8> 2>>  -->  Sets bit 1 in Flags 2 in
                                  header (force monospace)
```

PUTB

```
<PUTB table offset value>
```

Zapf syntax	**Inform syntax**
PUTB	storeb

All versions

Put a byte `value` in `table` at byte position `offset`. Actual address is table-address+offset.

Note that `table` can be a byte-address in dynamic memory.

Example:

```
<PUTB ,MYTABLE 1 !\A>        -->  Stores character A at
                                  position 1 in MYTABLE
```

PUTP

```
<PUTP object property value>
```

Zapf syntax	**Inform syntax**
PUTP	put_prop

All versions

Put `value` into `property` on `object`.

Example:

```
<OBJECT MYOBJ (MYPROP 123)>

<PUTP ,MYOBJ ,P?MYPROP 456>  -->  Stores 456 in property
                                  MYPROP on MYOBJ
```

QUIT

```
<QUIT>
```

Zapf syntax	**Inform syntax**
QUIT	quit

All versions

Halts game execution. No questions asked.

RANDOM

```
<RANDOM range>
```

Zapf syntax	**Inform syntax**
RANDOM	random

All versions

Returns random number between 1 and `range`. If `range` is negative the randomizer is reseeded with -`range` (absolut value of `range`).

Example:

```
<- <RANDOM 101> 1>  -->  Generates random number between 0-100
```

READ

```
<READ text parse>                          ;"Versions 1-3"
<READ text parse [time] [routine]>         ;"Version 4"
<READ text [parse] [time] [routine]>       ;"Versions 5-"
```

Zapf syntax	**Inform syntax**
READ	aread / sread

All versions

Reads text from keyboard and parse it. Result is stored in two byte-tables. Byte 0 in `text` most contain the max-size of the buffer and if `parse` is supplied, byte 0 of it most cointain max number of words that will be parsed.

After `READ`, `text` contains:

Byte	0	Max number of chars read into the buffer
	1	Actual number of chars read into the buffer
	2-	The typed chars all converted to lowercase

`parse` contains:

Byte	0	Max number of words parsed
	1	Actual number of words parsed
	2-3	Adress to first word in dictionary (0 if word is not in it)
	4	Length of first word
	5	Start position (in `text`) of first word
	6-9	Second word
	...	

Example:

```
<GLOBAL READBUF <ITABLE BYTE 63>>
<GLOBAL PARSEBUF <ITABLE BYTE 28>>
<ROUTINE READ-TEST ("AUX" WORDS WLEN WSTART WEND)
     <PUTB ,READBUF 0 60>
     <PUTB ,PARSEBUF 0 6>
     <READ ,READBUF ,PARSEBUF>
     <SET WORDS <GETB ,PARSEBUF 1>>  ;"# of parsed words"
     <DO (I 1 .WORDS)
          <SET WLEN <GETB .PARSEBUF <* .I 4>>>
          <SET WSTART <GETB .PARSEBUF <+<* .I 4> 1>>>
          <SET WEND <+ .WSTART <- .WLEN 1>>>
```

```
            <TELL "word " N .I " is " N .WLEN " char long. ">
            <TELL "The word is '">
            <DO (J .WSTART .WEND)
                 <PRINTC <GETB .READBUF .J>> ;"To lcase!"
            >
            <TELL "'." CR>
       >
>
```

See *The Inform Designer's Manual* (ch. §2.5, p. 44-46) for more details about `READ`.

REMOVE

```
<REMOVE object>
```

Zapf syntax	**Inform syntax**
REMOVE	remove_obj

All versions

Remove `object` from parent. See `MOVE` how to reattach it to another object.

Example:

```
<OBJECT ANIMAL>
<OBJECT CAT (LOC ANIMAL)>

<REMOVE ,CAT>       -->  Detach CAT from ANIMAL
```

REPEAT

```
<REPEAT (bindings...) expressions...>
```

REST

```
<REST table [bytes]>
```

RESTART

```
<RESTART>
```

Zapf syntax	**Inform syntax**
RESTART	restart

All versions

Restarts game. No questions asked. The only things that survives a restart are bit 0 and bit 1 of Flags 2 in header (setting for transcribing and monospace).

RESTORE

```
<RESTORE>                                ;"Versions 1-4"
<RESTORE [table] [bytes] [filename]>     ;"Versions 5-"
```

Zapf syntax	**Inform syntax**
RESTORE	restore

```
All versions
```

Restores a game to a previously saved state. All questions about filename and path are asked by the interpreter.

If RESTORE fails game execution continues with next statement after RESTORE.

If RESTORE is successful game execution continues from where the SAVE was issued (SAVE returns 2 in this case).

See *The Inform Designer's Manual* (ch. §42, p. 319) and *The Z-machine Standards Document* for a description about how to SAVE and RESTORE auxiliary files.

Example:

```
<ROUTINE SAVE-GAME ("AUX" RESULT)
     <SET RESULT <SAVE>>
     <COND (<=? .RESULT 0> <TELL "Save failed." CR>)>
     <COND (<=? .RESULT 1> <TELL "Save successful." CR>)>
     <COND (<=? .RESULT 2> <TELL "Restore successful." CR>)>
>

<ROUTINE RESTORE-GAME ()
     <RESTORE>
     <TELL "Restore failed." CR>
>
```

RETURN

```
<RETURN [value] [activation]>
```

Zapf syntax	**Inform syntax**
RETURN	ret

```
All versions
```

RETURN from current routine with value. Returns 1 (true) if no value is given.

RETURN is also used in commands that control program flow to exit program blocks. Also see AGAIN, BIND, DO, PROG and REPEAT for details how to control program flow.

Examples:

```
<RETURN>            --> Returns 1
<RETURN 42>         --> Returns 42
```

RFALSE

```
<RFALSE>
```

Zapf syntax	**Inform syntax**
RFALSE	rfalse

RFATAL

```
<RFATAL>
```

RSTACK

<RSTACK>

Zapf syntax	Inform syntax
RSTACK	ret_popped

RFALSE

<RTRUE>

Zapf syntax	Inform syntax
RTRUE	rtrue

SAVE

<SAVE [table] [bytes] [filename]>

Zapf syntax	Inform syntax
SAVE	save

SCREEN

<SCREEN window-number>

Zapf syntax	Inform syntax
SCREEN	set_window

SCROLL

<SCROLL window-number pixels>

Zapf syntax	Inform syntax
SCROLL	scroll_window

SET

<SET name value>

Zapf syntax	Inform syntax
SET	store

SETG

<SETG name value>

SOUND

<SOUND number [effect] [volume] [routine]>

Zapf syntax	Inform syntax
SOUND	sound_effect

SPLIT

<SPLIT number>

Zapf syntax	Inform syntax
SPLIT	split_window

T?

```
<T? expression>
```

TABLE

```
<TABLE [(table-flags...)] values...>
```

TELL

```
<TELL token-commands>
```

THROW

```
<THROW value stack-frame>
```

Zapf syntax	Inform syntax
THROW	throw

Versions: 5-

Used in conjunction with CATCH. THROW sets the stack to stack-frame and returns value (the result is that execution returns from the routine where the stack-frame were "caught" with value as the routines return value. Also see CATCH.

Example:

```
<ROUTINE TEST-CATCH ("AUX" X)
     <SET X <CATCH>>
     <THROWER .X>
     123
>

<ROUTINE THROWER (F)
     <THROW 456 .F>
>

<TEST-CATCH>   -->  456
```

USL

```
<USL>
```

Zapf syntax	Inform syntax
USL	show_status

VALUE

```
<VALUE name/number>
```

Zapf syntax	Inform syntax
VALUE	load

VERIFY

```
<VERIFY>
```

Zapf syntax	Inform syntax
VERIFY	verify

VERSION?

```
<VERSION? (name/number expressions...)...>
```

WINATTR

```
<WINATTR window-number flags operation>
```

Zapf syntax	Inform syntax
WINATTR	window_style

WINGET

```
<WINGET window-number property>
```

Zapf syntax	Inform syntax
WINGET	get_wind_prop

WINPOS

```
<WINPOS window-number row column>
```

Zapf syntax	Inform syntax
WINPOS	move_window

WINPUT

```
<WINPUT window-number property value>
```

Zapf syntax	Inform syntax
WINPUT	put_wind_prop

WINSIZE

```
<WINSIZE window-number height width>
```

Zapf syntax	Inform syntax
WINSIZE	window_size

XPUSH

```
<XPUSH value stack>
```

Zapf syntax	Inform syntax
XPUSH	push_stack

ZWSTR

```
<ZWSTR src-table length offset dest-table>
```

Zapf syntax	Inform syntax
ZWSTR	encode_text

Appendix A: Other Z-machine OP-codes

These OP-codes don't have direct ZIL-equivalent (they are used to call routines and control program counter).

Sources:

The Z-Machine Standards Document, Graham Nelson

ZAPF syntax	Inform Syntax	Description (Z specifikations 1.0)
CALL1	call_1s	Executes routine() and stores resulting return value.
CALL2	call_2s	Executes routine(arg1) and stores resulting return value.
CALL	call_vs	The only call instruction in Version 3. It calls the routine with 0, 1, 2 or 3 arguments as supplied and stores the resulting return value. (When the address 0 is called as a routine, nothing happens and the return value is false.)
ICALL1	call_1n	Executes routine() and throws away result.
ICALL2	call_2n	Executes routine(arg1) and throws away result.
ICALL	call_vn	Like CALL, but throws away result.
IXCALL	call_vn2	CALL with a variable number (from 0 to 7) of arguments, then throw away the result. This (and call_vs2) uniquely have an extra byte of opcode types to specify the types of arguments 4 to 7. Note that it is legal to use these opcodes with fewer than 4 arguments (in which case the second byte of type information will just be $FF).
JUMP	jump	Jump (unconditionally) to the given label. (This is not a branch instruction and the operand is a 2-byte signed offset to apply to the program counter.) It is legal for this to jump into a different routine (which should not change the routine call state), although it is considered bad practice to do so and the Txd disassembler is confused by it.
NOOP	nop	Probably the official "no operation" instruction, which, appropriately, was never operated (in any of the Infocom datafiles): it may once have been a breakpoint.
XCALL	call_vs2	Like IXCALL, but stores resulting value.

Appendix B – Field-spec for header

The information here is mostly from *The Z-Machine Standards Document, Graham Nelson* and ZILF Source Code. See *The Z-Machine Standards Document* for a more detailed discussion. The field-spec is used in LOWCORE and LOWCORE-TABLE.

Ordinary header

Field-spec	Byte	Ver	R/W	Description
ZVERSION	0-1	1-	R	Byte 0 Version number
		1-3	-	Byte 1 Flag 1
			R	Bit 1: Status line type: 0=score/turns, 1=hh:mm
			R	Bit 2: Story file split over two discs
			R	Bit 3: Tandy-bit
			R	Bit 4: Status line not available
			R	Bit 5: Screen-splitting available
			R	Bit 6: Is a proportional font the default
		4-	-	*01 Flag 1
			R	Bit 0: Colors available
			R	Bit 1: Picture displaying available
			R	Bit 2: Bold available
			R	Bit 3: Italic available
			R	Bit 4: Monospace (fixed) font available
			R	Bit 5: Sound effects available
			R	Bit 7: Timed keyboard input available
ZORKID/RELEASEID	2-3	1-	R	Release number (word). Note: Tradiionaly in Infocom only 11 bits are used for release-id (binary and *3777*). That suggest that the higher 5 bits sometime was used or reserved for other information.
ENDLOD	4-5	1-	R	Base of high memory (byte address)
START	6-7	1-5	R	Initial value of program counter (byte address)
		6	R	Packed address of initial "main" routine
VOCAB	8-9	1-	R	Location of dictionary (byte address)
OBJECT	*10-11	1-	R	Location of object table (byte address)
GLOBALS	*12-13	1-	R	Location of global variables table(byte address)
PURBOT	*14-15	1-	R	Base of static memory (byte address)
FLAGS	*16-17	-	-	Flags 2:
		1-	R/W	Bit 0: Set when transcripting is on
		3-	R/W	Bit 1: Set to force printing in monospace font
		6-	R/W	Bit 2: Int sets to request screen redraw, game clears when it complies with this
		5-	R	Bit 3: If set, game wants to use pictures
		3	R	Bit 4: Amigs ver of "The Lurking Horror" sets this probably sound.
		5-	R	Bit 4: If set, game wants to use UNDO

		5-	R	Bit 5: If set, game wants to use mouse
		5-	R	Bit 6:If set, game wants to use colors
		5-	R	Bit 7: If set,gsme wants to use sound
		6	R	Bit 8: If set, gaame wants to use menu
SERIAL	18-19	3-	R	Serial number,YY-part
SERI1	20-21	3-	R	Serial number,MM-part
SERI2	22-23	3-	R	Serial number,DD-part
FWORDS	24-25	2-	R	Location of abbreviations table (byte adress)
PLENTH	26-27	3-	R	Length of file
PCHKSUM	28-29	3-	R	File checksum
INTWRD	30-31	4-	R	Interpreter numbera nd version
INTID	30	4-	R	Interpreter number
INTVER	31	4-	R	Interpreter version
SCRWRD	32-33	4-	R	Screen width and hight
SCRV	32	4-	R	Screen height(lines), 255 = infinite
SCRH	33	4-	R	Screen width (characters)
HWRD	34-35	5-	R	Screen width in units
VWRD	36-37	5-	R	Screen height in units
FWRD	38-39	-	R	Font width and height
	38	5	R	Font width in units (width of '0')
		6-	R	Font height in units
	39	5	R	Font height in units
		6-	R	Font width in units (width of '0')
LMRG / FOFF	40-41	5-	R	Routines offset (divided by 8)
RMRG / SOFF	42-43	5	R	Static strings offset(divided by 8)
CLRWRD	44-45	5-	R	Default background and foreground color
	44	5-	R	Default background color
	45	5-	R	Default foreground color
TCHARS	46-47	5-	R	Address of terminating characters table (bytes)
CRCNT	48-49	5	R/W	???
TWID	48-49	6-	R	Total width in pixels of text sent to output stream 3
CRFUNC /STDREV	50-51	1-	R/W	Standard revision number
CHRSET	52-53	5-	R	Alphabet table address (bytes), or 0 for default
EXTAB	54-55	5-	R	Header extensiontable address (bytes)

Extended header

Field-spec	Byte	Ver	R/W	Description
	0-1	-	R	Number of further words in table
MSLOCX	2-3	5-	R	X-coordinate of mouse after a click
MSLOCY	4-5	5-	R	Y-coordinate of mouse after a click
MSETBL / UNITBL	6-7	5-	R/W	Unicode translation table (optional)
MSEDIR / FLAGS3	8-9	5-	R/W	Flags 3: Bit 0: If set, game wants to use transparency
MSEINV / TRUFGC	10-11	5-	R/W	True default foreground colour
MSEVRB / TRUBGC	12-13	5-	R/W	True default background colour
MSEWRD	14-15	5-	R/W	
BUTTON	16-17	5-	R/W	
JOYSTICK	18-19	5-	R/W	
BSTAT	20-21	5-	R/W	
JSTAT	22-23	5-	R/W	