Ozmoo

Ozmoo

A Z-machine interpreter for the Commodore 64 and similar computers

Release 5: 20 November 2020

Ozmoo was conceived and designed by Johan Berntsson and Fredrik Ramsberg. Special thanks to howtophil, Retrofan, Paul van der Laan, Jason Compton, Alessandro Morgantini, Thomas Bøvith, Eric Sherman, Paul Gardner-Stephen and Steve Flintham for testing, code contributions and bug fixes.

Contents

1 Program and Data Structures **3**
- Overview . . . 3
- Source files . . . 5
- Startup . . . 6
- Z-machine . . . 6
- Disk I/O . . . 6
- REU . . . 7
- Save and Restore . . . 7

2 Screenkernal **8**

3 Virtual Memory **9**
- The basics . . . 9
- Timestamps . . . 10
- The quick index . . . 11
- Efficient access to vmap . . . 12
- Banking . . . 13

4 Accented Characters **14**

5 Compiler Flags **16**
- General flags . . . 16
- Debug flags . . . 17

6 Floppy Configuration **19**

7 Printer Support **21**

8 Runtime Errors **22**

Chapter 1

Program and Data Structures

Overview

The main parts of Ozmoo are as follows:

- The interpreter This is the main program that sets up the memory structures, reads the header and dynmem part of the story file into memory, and then starts executing the program instruction by instruction in the Z-machine emulator, taking care to read memory from disk if addressing memory that doesn't fit in the computers RAM.
- Virtual memory buffers if virtual memory used. On some computers it is hard to use all the available RAM without time-consuming banking operations. To improve efficiency the virtual memory buffers are used when Ozmoo is compiled with virtual memory support, and located in RAM that is guaranteed to be accessible. By having the buffers in accessible RAM most operations can be done with direct RAM accesss, and banking is only needed when the program counter leaves the buffer.
- Z-machine stack This is used to store arguments and function calls.
- Virtual memory The rest of the available RAM is divided into 512-byte blocks and used to store parts of the story file as needed by the game state. The virtual memory is a complex topic which is covered in more detail in the next chapter.

The next figure shows how memory may be allocated when playing a game on the Commodore 64.

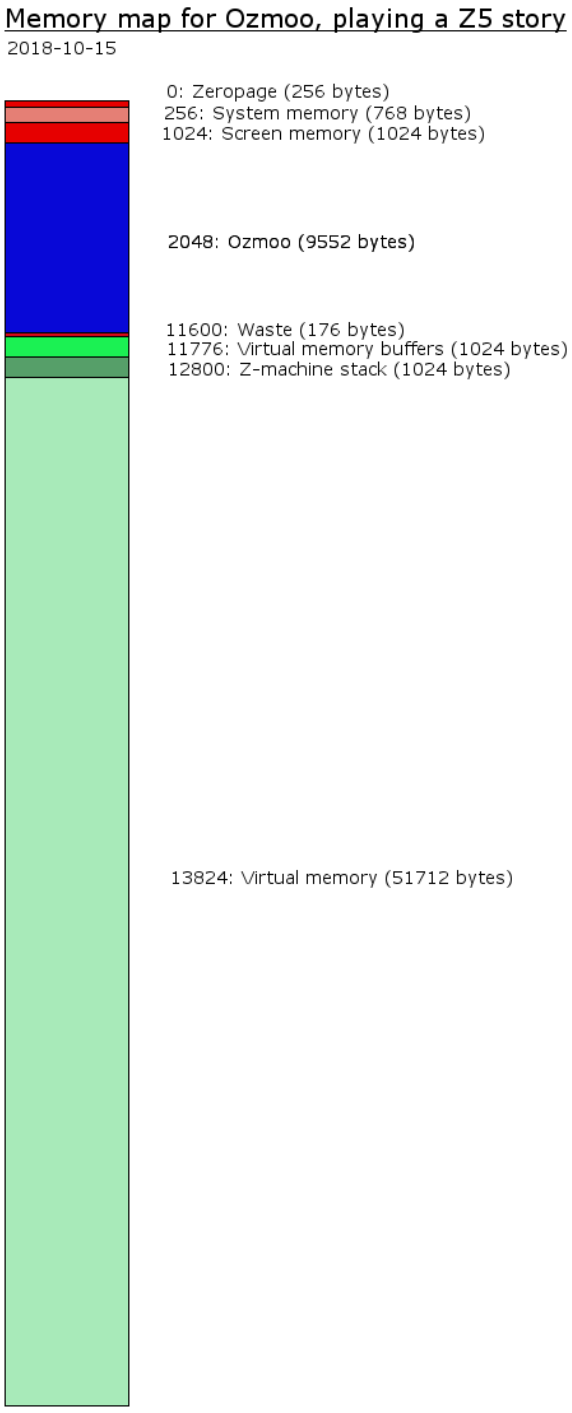

Figure 1.1: C64 memory overview

Source files

These source files are located in the asm dictory. The table also shows the most importand routines included in each source file.

File/Routine	Comment
ozmoo.asm	Ozmoo's main loop
initialise	the start of initialization
zmachine.asm	implements the Z-machine
constants.asm	zero page allocations and kernal labels
constants-c128.asm	C128 zero page allocations and kernal labels
constants-header.asm	labels to access the header part of a story file
disk.asm	read and write to floppy disk
readblock	read a memory block from the story file
readblocks	read a memory block from the story file
read_track_sector	read single sector from the floppy disk
dictionary.asm	routines to access the dictionary of a story file
objecttable.asm	routines to access the object tree defined in a story file
memory.asm	routines to access the Z-machine memory
picloader.asm	show a drawing while reading the story file
reu.asm	implements the Ram Expansion Unit for C64 and C128
screen.asm	printing routines
screenkernal.asm	low-level display routines
splashlines.asm	generated by make.rb. Contains splash screen text
splashscreen.asm	show the splash screen
stack.asm	implements the Z-machine stack
streams.asm	implements text streams for the Z-machine

File/Routine	Comment
text.asm	routines to read and write text
vdc.asm	low level routines to write text on the C128 in 80 columns mode
vmem.asm	virtual memory routines
zaddress.asm	implemented the Z-machine program counter
utilities.asm	various utilities
c65toc64wrapper.asm	wrapper file for Mega65 version. It makes sure that Ozmoo runs in enhanced C64 mode

Startup

Ozmoo programs are compressed with Exomizer into a file called Story, which is loaded and run like a Basic program. When Story is executed, the uncompressed Ozmoo program replaces Story and execution starts from the program_start label.

First the screen is initialised and the splash screen displayed (if any). Then the first part of the story file, containing the header and the dynmem part, is read. Optionally the rest of preloaded virtual memory is also read, and then the Z-machine program counter is set to the start of the story, as specified in the header.

Z-machine

The Z-machine executes instructions, using the Z-machine stack to keep track of calls and arguments.

Disk I/O

Ozmoo is designed to use the same floppy disc layout used by Infocom, with the first tracks and sectors being allocated to store the story file, and the remaining space on the floppy used to store the interpreter in the Story file.

The disk I/O routines are located in disk.asm. The virtual memory uses read_block and read_blocks to read data from the story file when needed.

There is read_track_sector which for reading a single sector of the story file specified by track and sector.

REU

If the computer detects a RAM Expansion Unit (REU), then it will be used to improve virtual memory performance. If a REU is detected, then once the dynamic memory has been read, the rest of the story file will be read into the REU. When the virtual memory need to read data from the story file by calling read_block or read_blocks (disk.asm) it will check and use the REU instead of reading sectors from the floppy drive.

Save and Restore

The story file is read piece by piece by mapping the program counter to the correct track and sector, and read it into memory. The main function doing this is readblocks located in disk.asm

disk.asm also contains save and restore functionality. The main functions are do_save and do_restore. The save files are normal files that contain some important internal variables such as the program counter, the Z-machine stack, and the dynmem part of the RAM.

Chapter 2

Screenkernal

Screenkernal, implemented in screenkernal.asm, is a replacement for the low-level screen output routines on the Commodore computers. It is needed to abstract away low-level implementation details so that new targets can be more easily added, and also to support custom text scrolling to to efficiently enable status lines, especially big ones used in games such as Border Zone and Nord and Bert. The number of lines to protect when scrolling is stored in window_start_row

The main functions of screenkernal are s_printchar, which replaced CHROUT $FFD2, and s_plot which replaced PLOT $FFF0. It also provides s_set_textcolour, s_delete_cursor, s_erase_window s_erase_line, s_erase_line_from_cursor and s_init.

Screenkernal is started by calling s_init. Internally it keeps track of the cursor position so it can put a character on the screen when s_printchar is called.

For the Commodore 64 version the characters are stored directly in the video memory, and the colour in the colour memory. For the Mega65 version, the only difference is that the screen is 80 characters wide which maes the video ram double size, but the colour memory has only space for 40 characters. To solve this the Mega65 version temporarily banks extra colour memory in place when printing a characters, and then removes it afterwards.

The Commodore 128 version detects if 40 or 80 columns mode is used while running the program. If 40 characters are used, then it works like the Commodore 64 version. But if 80 columns are used, then the C128's Video Display Controller chip (VDC) is used. Instead of writing directly into the video memory, character output, scrolling and other screen commands are sent by VDC registers. The file vdc.asm contains functions that make this communication easier.

The Plus/4 and Commodore 128 in 80 column mode doesn't use the same palette as the Commodore 64. Mapping tables (plus4_vic_colours and vdc_vic_colours) are used to assign C64 colours to their closest equivalents on these platforms.

Chapter 3

Virtual Memory

This chapter is based on a document written by Steve Flintham.

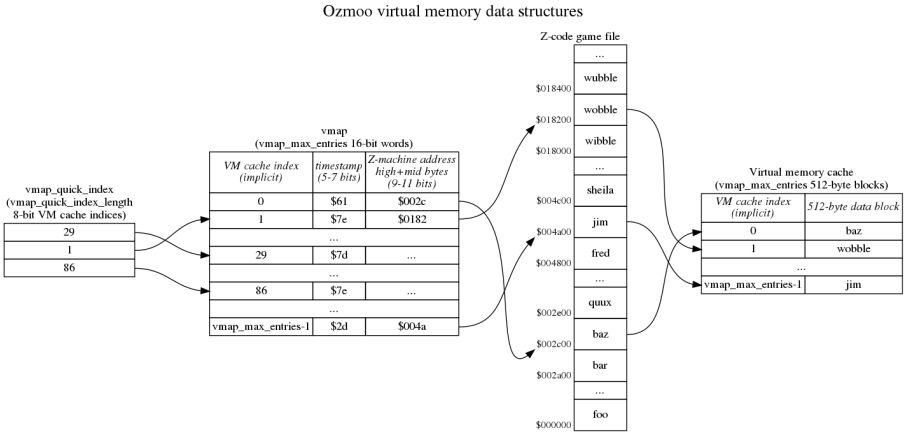

Figure 3.1: Virtual memory overview

The basics

The virtual memory subsystem is only used for the game's static and high memory, which are read-only. The game's dynamic memory is always held entirely in RAM.

The virtual memory code does most of its work in the read_byte_at_z_address subroutine. (This can be seen in vmem.asm; there are multiple versions of this subroutine in the file because conditional assembly is used to allow non-VM builds, but the VM version starts about halfway down.) It's entered with a

24-bit Z-machine address in A, X and Y (high byte in A, middle byte in X, low byte in Y) and returns with Y=0 and mempointer (a 2-byte zero page pointer) set so that "lda (mempointer),y" returns the byte at the 24-bit address given.

Note that read_byte_at_z_address isn't called for every single read from Z-machine high and static workspace; there are subroutines layered on top of it which only call it when necessary.

Virtual memory is handled in 512-byte blocks, which are always aligned at a 512-byte boundary in the game file and in memory. This means that every VM block has a Z-machine address of the form $abcd00, and the least significant bit of d is always 0.

The VM system has a cache with vmap_max_entries 512-byte blocks of RAM to use to hold blocks read from disc. There's a parallel data structure called the vmap with vmap_max_entries 16-bit words in to track what block of game data is in each cache block. At the most basic level, we can imagine that if block 4 of the cache contains the 512-byte block starting at $018200 in the game file, vmap[4] contains $0182.

If we want to access the byte at $018278, we round that down to the previous 512-byte boundary to get $018200 and then search through all the entries in vmap to see if any of them contain $0182. In this case, entry 4 does, so we'll set mempointer to cache_start_address+4*512+$078 and return. Where does the $078 come from? This is the low 9 bits of the address we were given, which can be though of as an offset to the byte of interest within this 512-byte block.

What if none of the vmap entries contains $0182? In that case we need to pick one to overwrite; let's say we pick entry 7. Because we only use virtual memory for read-only data, we can just read the 512-byte block at offset $018200 in the game file into memory at cache_start_address+7*512 (overwriting whatever was there), set vmap[7] to $0182 and return with mempointer set to cache_start_address+7*512+$078.

At the most basic level, that's all there is to it. But in practice there are some additional details we need to take care of.

Timestamps

We need to be able to make a sensible decision about which cache block we're going to overwrite when we need to read a block of data in from disc because it's not in the cache. What we want to do is keep hold of cache blocks we've recently used, and instead discard a block we haven't used in a while, on the reasonable assumption that we're more likely to use a block in the near future if we've recently used it.

To implement this, each vmap entry also contains a timestamp. There's a global "current time", called vmem_tick, and every time we access a cached block its

timestamp is set to vmem_tick. vmem_tick is incremented whenever we need to read data from disc because it's not in the cache. Note that several entries in the vmap can therefore share the same timestamp - if we access cache blocks 4, 8 and 22 without needing to read anything from disc, all of those will have the same timestamp (the current value of vmem_tick).

Also, there's a mechanism to keep vmem_ticks from getting too big for the space available. For a z3 game, we have 7 bits for storing tick values for each vmem block. We start with the value 0 and we can just increase it until we reach 127. When we hit 128, we change the tick counter to 64, and we reduce the tick value in all vmem blocks by 64. Since we're using unsigned numbers here, a block which had the tick value 27 now gets 0. A block which had tick value 100 now gets 36 etc. The next block we read from disk gets the tick value 64, the next after that gets 65 etc. Next time we reach 128, we just repeat the above procedure.

With that infrastructure in place, when we need to overwrite a block in the cache with a new block from disc, we can pick a block with the oldest timestamp. (Not *the* block with the oldest timestamp, because as noted above several blocks can share the same timestamp.) There's one caveat, which is that if the Z-machine program counter is currently pointing into a cache block, it is exempt from being overwritten - it would obviously be a bad thing to overwrite the instructions currently being executed with some arbitrary data! I won't go into too much detail on this here, because it's probably best discussed in the context of the routines layered on top of read_byte_at_z_address.

The timestamps are packed into the high bits of the 16-bit entries in vmap, with the low bits representing the high and mid bytes of the Z-machine address. For Z3 games, where Z-machine addresses only have 17 bits (a maximum game size of 128K), only 9 bits of the vmap entry are needed for the high and mid bytes of the address and 7 bits are available for the timestamp. Larger versions of the Z-machine need more bits for the high and mid bytes so fewer bits are available for the timestamp; a Z8 game (with 19 bit addresses for a maximum game size of 512K) only has 5 bits available for the timestamp. This limited timestamp resolution is probably why vmem_tick is only incremented when a block needs to be read from disc, not every time read_byte_from_z_address is called.

vmem_tick is actually held in memory "pre-shifted" for convenience of using it to compare or update the high byte of the 16-bit entries in vmap, so rather than incrementing by 1 at a time, it increments by 2 at a time for Z3 games, 8 at a time for Z8 games and 4 for everything else. (In the code, the increment is a constant called vmem_tick_increment.)

The quick index

In general we have to do a linear search of vmap in order to see if a particular 512-byte block of the game is already in RAM (and where in RAM it is, if it's in

RAM). The vmap might contain as many as 255 entries (if we have huge amounts of sideways RAM), and it's likely to contain at least 64 entries, representing 32K of cache, so this is potentially quite slow.

It's quite likely that there's a relatively small "working set" of 512-byte blocks which we're going to be accessing over and over again. For example, maybe one 512-byte block contains a Z-machine function with a loop in, and inside that loop we call another Z-machine function which lives in a separate 512-byte block.

We therefore maintain a "quick index" containing the cache indices of the blocks we've accessed most recently. (There are vmap_quick_index_length entries in this list, which in practice means there are 6 entries.) We look at the corresponding entries in vmap first to see if those entries have the 512-byte block we're interested in, and if they do we can avoid doing the full linear search. Whenever we have to do the full vmap search, we overwrite the oldest entry in the quick index with the index we found from the full search. The quick index is effectively a circular buffer of vmap_quick_index_length entries, with vmap_next_quick_index pointing to the oldest entry.

As a consequence of this, the vmap entries pointed to by the quick index are those with the most recent timestamps.

Efficient access to vmap

Each vmap entry is a 16-bit word, so the obvious way to store vmap would be:

```
    low byte of entry 0
    high byte of entry 0
    low byte of entry 1
    high byte of entry 1
    ...
```

The disadvantage of this is that when we're using index registers to step through vmap, we need to do double increment (or decrement) operations:

```
    ldy #0
.loop
    lda vmap,y ; get low byte of entry
    lda vmap+1,y ; get high byte of entry
    ...
    iny
    iny
    cpy #max_entries*2
    bne loop
```

We don't actually need the two bytes to be adjacent in memory, and so instead we have two separate tables. vmap_z_l stores the low bytes:

```
    low byte of entry 0
```

```
    low byte of entry 1
    ...
```

and vmap_z_h stores the high bytes in the same way.

With this arrangement, we don't need double increments (or decrements) to step through the table:

```
    ldy #0
.loop
    lda vmap_z_l,y ; get low byte of entry
    lda vmap_z_h,y ; get high byte of entry
    ...
    iny
    cpy #max_entries
    bne loop
```

This also has the advantage that we can access up to 256 entries using our 8-bit index registers, instead of 128 entries if we had to do double increment/decrement. The Commodore 64 version of Ozmoo doesn't need this, but the Acorn sideways RAM version benefits from it and there are a couple of minor tweaks to the code to make this work. (If you know there are <128 entries, you can use dex:bpl to control looping over the table and avoid needing a cpx #255 to detect wrapping.)

Banking

On the Commodore 64 there's an additional wrinkle because some blocks of RAM are hidden behind the kernal ROM and there's a mechanism to copy those blocks of RAM into cache blocks in always-visible RAM when necessary.

The first_banked_memory_page variable stored the high byte of the first 512 block of RAM that isn't always visible. On the C64 this is $d0, since the I/O registers are located from $d000, and unrestricted reading/writing to these memory position cause all kinds of trouble.

When read_byte_at_z_address detects that we want to read data from a block under the non-accessible memory ($d000-ffff) then we will swap one of the blocks in the always-visible RAM with the non-accessible block. This is done by copy_page (in memory.asm), which can copy the file securely from any memory position using memory banking as needed.

Chapter 4

Accented Characters

Ozmoo has some support for using accented characters in games. Since the Commodore 64 doesn't really support accented characters, some tricks are needed to make this work.

The Z-machine, which we are emulating, uses ZSCII (an extended version of ASCII) to encode characters.

The Commodore 64 uses PETSCII (a modified version of ASCII) to encode characters. PETSCII has 256 different codes (actually less, since some code ranges are just ghosts, i.e. copies of other code ranges). Some PETSCII codes are control codes, like the ones to change the text colour or clear the screen. 128 of the PETSCII codes (plus some ghost codes) map to printable characters. A character set contains these, plus reverse-video versions of all 128, adding up to 256 characters all in all.

So, all in all there are basically 128 different characters, and that's it. They include letters A-Z, digits, special characters like !#$%& etc, some graphic characters, and then either lowercase a-z or 26 more graphic characters. There are no accented characters.

To build a game with Ozmoo with accented characters, we typically need to:

- Decide which (less important) characters in PETSCII and in the C64 character set will be replaced with the accented characters we need.
- Create a font (character set) where these replacements have been made
- Create mappings from PETSCII to ZSCII for when the player enters text
- Create mappings from ZSCII to PETSCII for when the game outputs text

There are already fonts in place for some languages (see documentation/fonts.txt). You can also supply your own font.

The character mappings are created at the beginning of the file streams.asm. There are already mappings for the languages which there are fonts for (see the

Fonts chapter above).

To build a game using accented characters, the make command may look like this:

```
ruby make.rb examples\Aventyr.z5 -f fonts\PXLfont-rf-sv.fnt -cm:sv
```

where

```
-f sets the font to use.
-cm sets the character mapping to use.
```

If you want to create a character mapping for say Czech, and you know you will never want to build a game in Swedish, you can just replace the Swedish mapping in streams.asm and use -cm:sv to refer to your Czech mapping.

The definition of ZSCII can be found at

https://www.inform-fiction.org/zmachine/standards/z1point1/sect03.html#eight

The accented characters which are available by default, and which Ozmoo can use, are in a table under 3.8.7.

The definition of PETSCII can be found at: http://sta.c64.org/cbm64petkey.html Please read the notes below the table as well.

If you compile a game in Debug mode (Uncomment the 'DEBUG' line near the start of make.rb), Ozmoo will print the hexadecimal ZSCII codes for all characters which it can't print. Thus, to create mappings for a game in a new language, you can start by running it in Debug mode to see the ZSCII character codes in use.

Chapter 5

Compiler Flags

The flags described here can be set on the Acme commandline using the syntax -D[flag]=1

General flags

`BGCOL=n`

Set the background colour.

`BORDERCOL=n`

Set the border colour.

`CACHE_PAGES=n`

Set the minimum number of memory pages to use for the cache (used to buffer pages otherwise residing at $D000-FFFF$).

`COL2=n`
`COL3=n`
`COL4=n`
`COL5=n`
`COL6=n`
`COL7=n`
`COL8=n`
`COL9=n`

Replace color in Z-machine palette with a certain colur from the C64 palette.

`CUSTOM_FONT`

Tell the interpreter that a custom font will be used.

`DANISH_CHARS`
`FRENCH_CHARS`
`GERMAN_CHARS`
`ITALIAN_CHARS`
`SPANISH_CHARS`
`SWEDISH_CHARS`

Map national characters in ZSCII to their PETSCII equivalents. No more than one of these can be enabled.

`SMALLBLOCK`

When using virtual memory, use a blocksize of 512 bytes rather than 1024 bytes.

`STACK_PAGES=n`

Set the number of memory pages to use for stack.

`STATCOL=n`

Set the statusline colour. (only for z3).

`VMEM`

Utilize virtual memory. Without this, the complete game must fit in C64 RAM available above the interpreter and below $D000, all in all about 40 KB. Also check section "Virtual memory flags" below.

`Z3`
`Z4`
`Z5`
`Z8`

Build the interpreter to run Z-machine version 3, 4, 5 or 8.

Debug flags

(If any of the flags in this section are enabled, DEBUG is automatically enabled too.)

`DEBUG`

Print debug information, print descriptive error messages, store a trace of the instructions which have been executed and print them in case of an exception. Also check section "Debug flags" below.

`BENCHMARK`

When the game starts, replay a number of colon-separated commands which have been stored with the interpreter (in the file text.asm). Charcode 255 in this command sequence means print the number of jiffies since the computer was started.

`COUNT_SWAPS`

Keep track of how many vmem block reads have been done.

`PREOPT`

Built the interpreter in PREOPT mode (used to pick which virtual memory blocks should be preloaded into memory when the game starts).

`PRINT_SWAPS`

Print information whenever a memory block is loaded into memory.

`TRACE`

Print trace information.

`TRACE_FLOPPY`

Print trace information for floppy operations.

`TRACE_FLOPPY_VERBOSE`

Print verbose trace information for floppy operations.

`TRACE_PRINT_ARRAYS`

?

`TRACE_READTEXT`

Print trace information for opcode read/aread/sread.

`TRACE_SHOW_DICT_ENTRIES`

?

`TRACE_TOKENISE`

Print trace information for the tokenization process.

`TRACE_VM`

Print trace information for the virtual memory system.

`VICE_TRACE`

Send the last instructions executed to Vice, to aid in debugging.

`VIEW_STACK_RECORDS`

Print whenever the stack size hits a new high, or when the number of bytes pushed onto the stack within a frame reaches a new high.

Chapter 6

Floppy Configuration

Interpreter configuration is stored in two blocks on the boot disk (track 19, sector 0-1). The interpreter loads this information when it starts.

Contents:

```
4 bytes: Game signature: A 32-bit random number.

--- Disk information block ---
1 byte: Number of bytes used for disk information, including this byte
1 byte: Interleave (in range 1-21)
1 byte: Number of save slots which can fit on a disk (in range 4-132)
1 byte: Number of disks used (disk 0 is the save disk(s),
        disk 1 .. are game disks
For each disk:
        1 byte: n: Number of bytes used for this entry, including this byte.
        1 byte: Device#. Should be 8,9,10,11 or 0, meaning it's not decided
                         yet - it's up to the terp to set the device#.
        2 bytes: Last story block # + 1 (high endian) on this disk
        1 byte: t: Number of tracks used for story data
                   (If this number is > 0, this is a story disk!)
        t bytes: Sectors used for story data:
                 Bit 0-4: # of sectors used.
                 Bit 5-7: First sector# used.
                 (Example: %01010000: Use 16 sectors, starting with #2)
        x bytes: Disk name in Petscii. Special codes:
            0: End of string
            128: "Boot "
            129: "Story "
            130: "Save "
            131: "disk "
```

```
--- Vmem information block ---
1 byte: Number of bytes used for vmem information, including this byte
1 byte: Number of vmem blocks suggested for initial caching
1 byte: Number of vmem blocks already preloaded
x bytes: vmap_z_h contents for the suggested blocks
x bytes: vmap_z_l contents for the suggested blocks
```

Typical size for a 3-disk (+ save disk) game:

4 + 1 + 1 + 1 + 1 + 2 * (1 + 1 + 2 + 1 + 40 + 5) + 2 * (1 + 1 + 2 + 1 + 0 + 3) + 1 + 1 + 1 + 100 + 100 = 8 + 2 * 50 + 2 * 8 + 203 = 8 + 100 + 16 + 203 = 327 bytes

An interpreter needs to reserve this space for disk information:

z3: 1 + 1 + 1 + 1 + 2 * (1 + 1 + 2 + 1 + 0 + 3) + (1 + 1 + 2 + 1 + 40 + 6) = 4 + 2 * 8 + 51 = 71 bytes

z4/z5/z8: 1 + 1 + 1 + 1 + 2 * (1 + 1 + 2 + 1 + 0 + 3) + 2 * (1 + 1 + 2 + 1 + 40 + 5) = 4 + 2 * 8 + 2 * 50 = 120 bytes

1581 drive support: This system should work, without extending the limits above, using only 31 sectors per track for story data, while allowing z8 games of up to 512 KB in size on a single disk. If we want to store several games on a single 1581 disk, we should add a track offset in each disk entry (i.e. saying that tracks below track x are considered empty).

Possible 1571 support:

- z3 games: No change
- z4/z5 games: A single disk using only track 1-53 can hold all story data. Disk information will then fit in less than the number of bytes stated above.
- z8 games: Disk information for a single drive game will fit in the number of bytes stated above. Double 1571 drive support is possible but not a high priority.

Chapter 7

Printer Support

Currently Ozmoo doesn't have any support for printers, but hooks exist that could be used to add such functionality.

Infocom provided printer output through the transcript command, which redirected text output to output stream 2. stream.asm implements streaming, and stream_print_output can be extended to handle stream 2 and send the bytes to the printer.

We currently have no hardware to test printer support on, and no immediate plans to provide such support. However, we are happy to accept patches.

Chapter 8

Runtime Errors

These are the runtime errors that may occur when running Ozmoo:

- ERROR_UNSUPPORTED_STREAM = 1

The Z-machine supports certain streams for input and output of text. If a program tries to open a stream number which is not defined or which is not supported by Ozmoo, this error occurs.

- ERROR_CONFIG = 2

The boot disk of a program built with Ozmoo has config information on sector 1:0 and 1:1 (unless it's built as a single file program). This error means there seems to be something wrong with this information. The information is copied to memory when the game boots and then used whenever a block of game data needs to be copied from disk, so the problems may be discovered when the boot disk is no longer in the drive.

- ERROR_STREAM_NESTING_ERROR = 3

The Z-machine has a stack for memory streams. If the program tries to pull items from this stack when it is empty, this is the resulting error.

- ERROR_FLOPPY_READ_ERROR = 4

There was a problem reading from the disk.

- ERROR_STACK_FULL = 6

The program tried to make a routine call or push data onto the stack when there was not enough room. This means there is a bug in the program, or it needs a bigger stack. Stack size can be set with a commandline parameter to make.rb.

- ERROR_STACK_EMPTY = 7

The program tried to pull a value from the stack when it was empty.

- ERROR_OPCODE_NOT_IMPLEMENTED = 8

An unknown Z-machine opcode was encountered. This can happen if the wrong disk is in the drive when Ozmoo tries to retrieve a block of program code.

- ERROR_USED_NONEXISTENT_LOCAL_VAR = 9

Each routine in Z-code has between 0 and 15 local variables. If an instruction references a local variable number which is not present in this routine, this is what happens. Normally, a compiler like Inform doesn't let the programmer write code which can cause this, unless you skip the highlevel language and write Z-machine assembler.

- ERROR_BAD_PROPERTY_LENGTH = 10

The program tried to use an object property of an illegal length, where a property value has to be one or two bytes.

- ERROR_UNSUPPORTED_STORY_VERSION = 11

The first byte of the story file must match the Z-machine version for which the interpreter was built (3, 4, 5 or 8). Otherwise, this occurs.

- ERROR_OUT_OF_MEMORY = 12

The program referenced memory which is higher than the last address in the story file.

- ERROR_WRITE_ABOVE_DYNMEM = 13

The program tried to write to memory which is not part of dynamic (RAM) memory.

- ERROR_TOO_MANY_TERMINATORS = 15

A dictionary in a Z-machine program holds a list of terminating characters, which are used to separate words. It is illegal for this list to hold more than ten characters. If it does, this error occurs.

- ERROR_NO_VMEM_INDEX = 16

The vmem_oldest_index is only populated inside the loop if we find a non-PC vmem block older than the initial vmem_oldest_age of $ff. That should always happen, but to assert that the code is correct the debug version of Ozmoo checks that vmem_oldest_index is valid and issues ERROR_NO_VMEM_INDEX if not.

- ERROR_DIVISION_BY_ZERO = 17

An attempt was made to divide a number by zero.