Ozmoo

Ozmoo

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

Release 9: 28 November 2021

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, Steve Flintham, Bart van Leeuwen and Karol Stasiak for testing, code contributions and bug fixes.

Contents

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 config into memory, and then starts executing the program instruction by instruction in the Z-machine emulator, taking care to read portions of the Z-code file from disk as needed.
- Virtual memory buffers 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. The stack is described in detail in a chapter of its own.
- 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 a chapter of its own.

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

Source files

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

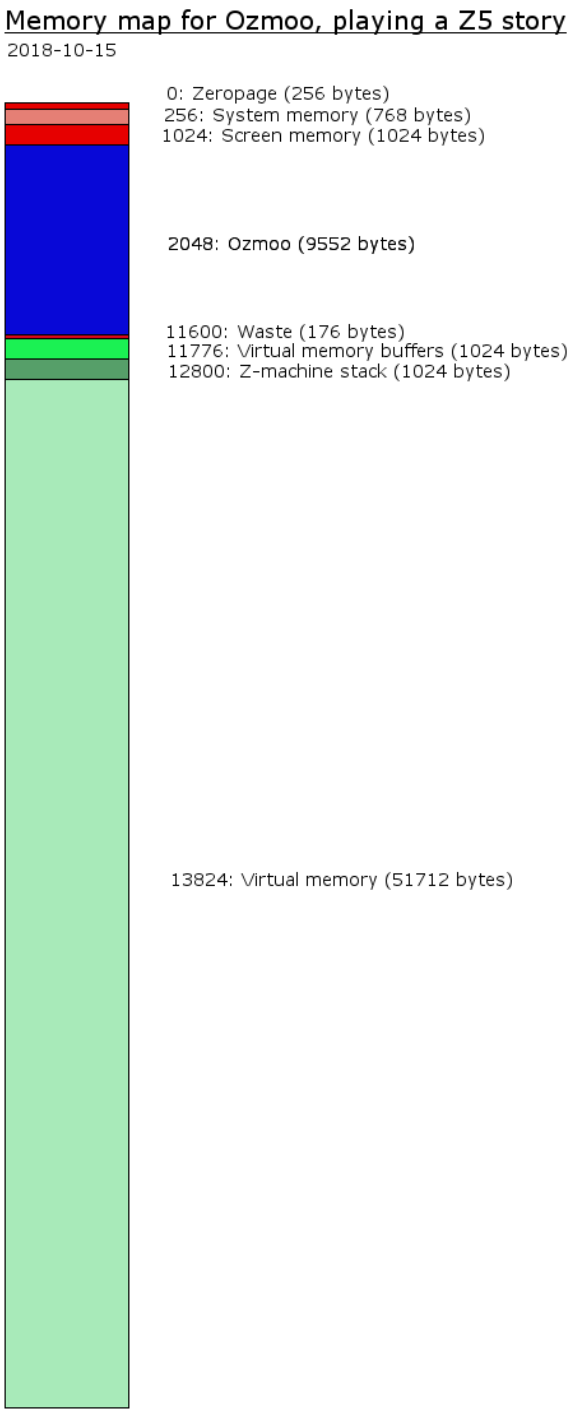

Figure 1: C64 memory overview

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 interface for C64 and C128
screen.asm	printing routines
screenkernal.asm	low-level display routines
splashlines.asm	generated by make.rb. Contains splash screen text
splashlines.tpl	template used by make.rb to generate splashlines.asm
splashscreen.asm	show the splash screen
stack.asm	implements the Z-machine stack
streams.asm	implements text streams for the Z-machine
text.asm	routines to read and write text

File/Routine	Comment
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

Startup

Ozmoo programs are compressed with Exomizer into a file called `story`, which is loaded and run like a Basic program. This file is sometimes referred to as a boot file. When this program 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 config blocks (the first two blocks on the boot disk) are read, Ozmoo checks for an REU, the interpreter may ask the player to insert the story disk(s) into the drive(s), Ozmoo may read all of the story file into the REU, or it may load parts of it into RAM (if there is still some empty RAM after uncompressing the boot file). Then the Z-machine program counter is set to the start of the story, as specified in the header, and execution of the Z-machine program begins.

Z-machine

The Z-machine executes instructions, using the Z-machine stack to keep track of calls and arguments. A Z-code file (AKA game file or story file, like zork1.dat or planetfall.z3) is essentially a snapshot of the memory of a virtual computer (the Z-machine). A Z-code file is divided into dynamic memory (up to 64 KB) and static memory (everthing above dynamic memory, up to 512 KB). Dynamic memory is the only part that can be altered during program execution. Ozmoo keeps the entire contents of dynamic memory in RAM at all times, while using whatever RAM is left for virtual memory, to hold as much of static memory as possible. Whenever the player moves into a new location or tries something new and the disk starts spinning, this is because the program needs a part of static memory that isn't in RAM, so it has to be fetched from disk. What's so nice about this system is that it's handled entirely by the interpreter. The author of a game doesn't have to figure out when to load which parts of the story file - they can just write their game as if it's going to run on a computer with 128 or

256 KB of RAM, and the interpreter makes it work on a computer with much less RAM than that.

To further conserve space, the Z-machine has a text compression scheme built in. Basically it means each character is translated into one or more five-bit codes. By default, the space character and all lowercase letters fit into a single five-bit code each, while uppercase letters and special characters require two or four five-bit codes. These five bit codes are then grouped together so three of them occupy two bytes. I.e. “adventure” is stored as “adv” in two bytes, “ent” in two bytes and “ure” in two bytes, thus using six bytes. On top of this, there is a system of abbreviations, so common strings can be replaced by two five-bit codes.

Disk I/O

Ozmoo is designed to use a floppy disk layout similar to what Infocom used, but it has some extra options such as using multiple disk drives to play bigger games. Config information for the game is stored in sector 0 and 1 of track 1 on the boot disk. The static memory portion of the story file is stored in the track 1 and up on the story disk(s). The boot file, containing the interpreter, dynamic memory and as much of static memory as possible, is stored on the boot disk or in the remaining free space of the combined boot / story disk.

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 a routine read_track_sector for reading a single sector of the story file specified by track number and sector number.

REU

If Ozmoo detects a RAM Expansion Unit (REU), the player gets the question if they want to use it. If they do, the entire story file is read into the REU before the game starts. When the virtual memory system needs to read data from the story file by calling read_block or read_blocks (disk.asm) it will read the blocks from the REU instead of 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.

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. 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.

The stack

The exact layout of the stack is an implementation detail, not part of the Z-machine specification. However, the Z-machine specification does say a few things about how the stack should *work*. Each routine call creates a stack frame. Instructions can push any number of values onto the stack, and pull values from the stack, or read or change the top value without pushing or pulling it. A value is always word-sized. Pushed values can only be accessed by instructions in the same routine (= same stack frame). When a routine returns, all values which have been pushed inside that routine and are still on the stack are discarded. The stack resides outside the Z-machine, and there is no way for a Z-machine program to check if there are values on the stack, if there is room for more values, how many stack frames there are etc.

The stack in Ozmoo consists of a number of words. The default size is 512 words. With each procedure call that is made, a new stack frame is created. The first stack frame begins at the first address of the stack (label stack_start).

A stack frame consists of the following parts:

1. The current values of all local variables of the routine (0-15 words).
2. Frame header (2 words)

   Frame header A has the following parts:

   Byte 0, bit 7: 1 = The return value of this procedure call should be written into a variable

   Byte 0, bit 4-6: In version 5+, the number of parameters this procedure was called with (0-7)

   Byte 0: bit 0-3: The number of local variables this procedure has (0-15)

   Byte 1, bit 7: 1 = this routine call was done due to an input interrupt

   Byte 1, bit 0-2: Bit 16-18 of the return address

   Frame header B:

Bit 0-15 of the return address (highbyte first).

3. Pushed words (0 or more words)
4. Number of pushed bytes (1 word)

The first frame lacks part 1 and 2. The current frame lacks part 4.

Apart from this, Ozmoo also has a few variables on zeropage to keep track of the stack:

stack_ptr (word): a pointer to where part 3 of the current stack frame begins

stack_pushed_bytes (word): A counter of how many words part 3 of the current frame consists of.

stack_has_top_value (byte): If the value is 1, the last value pushed onto the stack in the current frame hasn't been put on the actual stack, but resides in the variable stack_top_value instead.

stack_top_value (word): If stack_has_top_value is 1, this contains the top value on the stack.

The stack_top_value variable has been added for performance reasons.

Frame 0 isn't a full stack frame. When a Z-machine interpreter begins execution of a program, it just starts at the start address given in the header of the story file, with an empty stack. The start address can be in the middle of a routine. This means the interpreter doesn't know of any local variables, and there is no return address - it's not legal to perform a return from this routine. This "fake" routine is just a simple way to get started, and the only thing you want to do in this routine is call a "real" routine and then perform quit. Inform 6 adds such a routine automatically, and it calls the routine "main" which must be defined in the Inform 6 program and then quits. To make things more confusing, this little "fake" routine is called "main" when disassembling the Z-code program.

Let's go through an example program and what it does to the stack. We run it until it performs @read_char and then we pause and look at the stack. This is a version 5 program.

Inform's built-in "fake" main routine (on address $4ec) calls our main routine (on address $514). This causes Omzoo to close the current frame (Frame 0, which isn't a full stack frame) by moving the top value from stack_top_value (if any) to the stack, and writing the total number of pushed bytes as a word to the stack. In this case, there are no values on the stack, and no value in stack_top_value, so Ozmoo just copies the value in stack_pushed_bytes (0) to the stack.

Ozmoo then starts a new stack frame - frame 1. The main routine at $514 has 0 local variables, so it doesn't reserve any space for part 1. It then writes the frame header to the stack. Frame header A is $80, $00 because the procedure call is made using call_vs which expects the return value to be written into

```
!% -~S
!% $OMIT_UNUSED_ROUTINES=1

[foo x y;
    @push 9;
    @push 3;
    print x;
    new_line;
    print "Press a key: ";
    @read_char y;
    print y;
];

[main;
    @push 7;
    foo(5);
];
```

Figure 2: stackdemo.inf, Inform 6 source code

```
Main routine 4ec, 0 locals

  4ed:  e0 3f 01 45 ff          CALL_VS         514 -> Gef
  4f2:  ba                      QUIT

Routine 4f4, 2 locals

  4f5:  e8 7f 09                PUSH            #09
  4f8:  e8 7f 03                PUSH            #03
  4fb:  e6 bf 01                PRINT_NUM       L00
  4fe:  bb                      NEW_LINE
  4ff:  b2 ...                  PRINT           "Press a key: "
  50a:  f6 ff 02                READ_CHAR       -> L01
  50d:  e6 bf 02                PRINT_NUM       L01
  510:  b0                      RTRUE

Routine 514, 0 locals

  515:  e8 7f 07                PUSH            #07
  518:  da 1f 01 3d 05          CALL_2N         4f4 (#05)
  51d:  b0                      RTRUE
```

Figure 3: stackdemo.zasm, disassembly

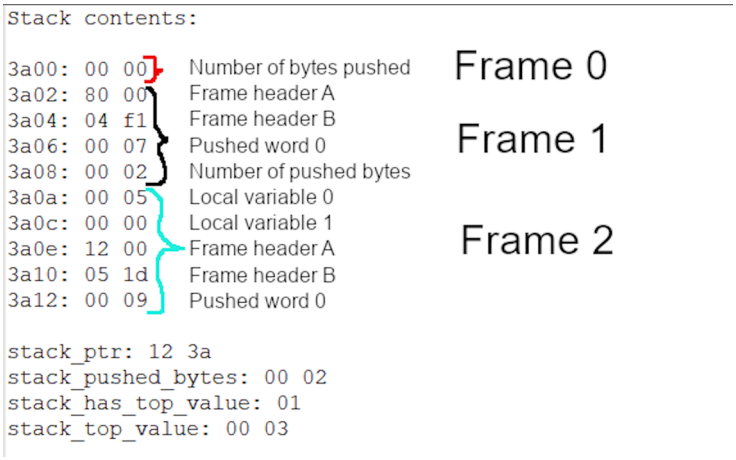

Figure 4: stackdemo, stack contents

a variable, there are no arguments and the routine has no local variables, the routine was not called due to an interrupt, and bit 16-18 in the return address are all 0. Frame header B holds the value $04f1, which points to the last byte of the call_vs statement, holding the variable number where the result is to be stored upon return.

Program execution now continues in our main routine at address $515 (the routine's official start address is 514, but the first instruction is on address $515.) The first instruction pushes the value 7 onto the stack. The second instructions performs a call_2n instruction to call the foo routine with one argument, which is the value 5. We can see that the value 7 has been pushed onto the stack, and that the counter says a total of 2 bytes have been pushed onto the stack in this frame.

Ozmoo now opens stack frame 2. The routine being called (the foo routine located at address $4f4) has two local variables, so Ozmoo makes room for two words on the stack for part 1 of the frame. The first variable gets the value 5, since the procedure was called with this as an argument. Ozmoo then adds part 2, the header. Frame header A gets the value $12, $00 because the routine was not called with an instruction which expects the return value to be written into a variable, the routine call was made with 1 parameter, the routine has 2 local variables, the routine was not called due to an interrupt, and bit 16-18 of the return address are 0. Frame header B holds the value $051d, which is where execution should resume upon return from this routine.

The foo routine pushes the value 9 and then the value 3. At the time when we inspect the stack, the value 3 is held in stack_top_value. If we were to push another value, or call a procedure, this value would be written to the stack as well.

There are two more print instructions and a new_line instruction before read_char, but none of them interfere with the stack. When we hit read_char, we paused the program and inspected the stack.

Virtual Memory

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

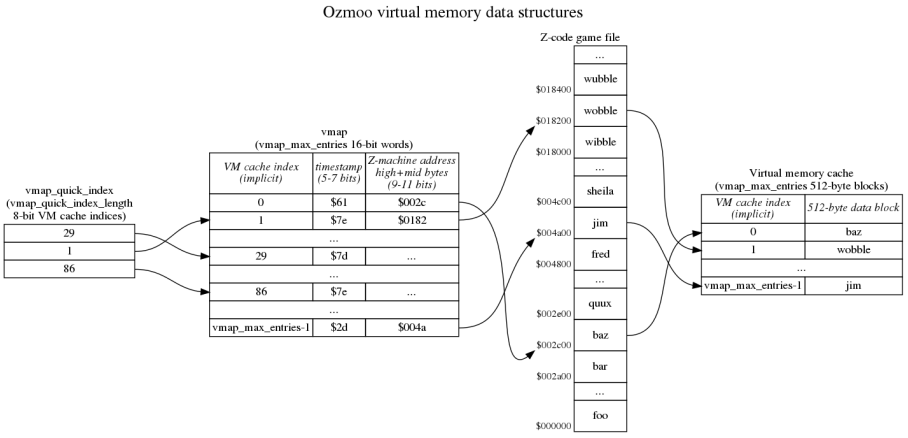

Figure 5: Virtual memory overview

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 8 bits for storing tick values for each vmem block. We start with the value 0 and we can just increase it until we reach 255. When we hit 256, we change the tick counter to 128, and we reduce the tick value in all vmem blocks by 128. Since we're using unsigned numbers here, a block which had the tick value 27 now gets 0. A block which had tick value 150 now gets 22 etc. The next block we read from disk gets the tick value 128, the next after that gets 129 etc. Next time we reach 256, 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. But note that the lowest bit of the address is always zero since the memory is divided into 512-byte blocks. Assuming this zero we can instead store the address in 8 bits and increase the time stamp resolution to 8 bits.

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 6 bits available for the timestamp. This limited timestamp resolution is the reason 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 always incrementing by 1 at a time, it increments by 1 at a time for Z3 games, 2 at a time for Z4 and Z5 games and 4 at a time Z8 games. (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 can contain up to 255 entries (depending on platform),

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, Commodore 128 and MEGA65 there's an additional wrinkle because some blocks of RAM are hidden behind the I/O area and the kernal ROM and there's a mechanism to copy those blocks of RAM into the so-called vmem cache 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 page in the non-accessible memory ($d000-ffff) then we will copy that page (256 bytes) from the non-accessible memory to one of the pages in the vmem buffer. This is done by copy_page (in memory.asm), which can copy a page securely from and to any memory position using memory banking as needed.

Sound

While several Infocom games would play high and low-pitched beeps, a few games had extended sound support using sample playback. Ozmoo supports the basic sound effects (beeps) on all platforms, and extended sounds on the MEGA65.

Sound support is implemented in sound.asm. Extended sounds also use sound-wav.asm to parse sample files stored in the WAV format. The WAV files need to be 8 bit, mono. Audacity can be used to export wav files in the correct format:

- Select "File/Export/Export as WAV" from the main menu
- Select "Other compressed files" as the file type
- Select "Unsigned 8-bit PCM" as the encoding
- Save the file

If extended sound is to be used, then make.rb must be called with the `-asw path` switch. If set, then all .wav files in the `path` will be added to the .d81 floppy created for the MEGA65, and the SOUND assembly flag will be set when building Ozmoo.

For legacy reason AIFF is also supported, using sound-aiff.asm, and the `-asa path` switch. Code has also been added to detect and handle sound effects in `The Lurking Horror` game from Infocom, which isn't standard compliant.

When compiled with extended sound support, Ozmoo will preload all sound files during startup into the MEGA65's attic memory, and then copy each sound effect into fast memory on demand when the `@sound_effect` command is used in the game code.

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 chapter on fonts in the Ozmoo manual).

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

```
ruby make.rb examples\Aventyr.z5 -f fonts\sv\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.

Note that in the mapping from PETSCII to ZSCII, you must also convert any accented characters from uppercase to lowercase, i.e. for Swedish PETSCII Ä is mapped to a ZSCII ä.

If you build a game using Ozmoo's 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.

Assembly Flags

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

General flags

`BGCOL=n`
`BGCOLDM=n`

Set the background colour for normal mode and darkmode.

`BORDERCOL=n`
`BORDERCOLDM=n`

Set the border colour for normal mode and darkmode.

`CACHE_PAGES=n`

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

`CURSORCOL=n`
`CURSORCOLDM=n`

Set the cursor colour for normal mode and darkmode.

`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.

`CONF_TRK=n`

Tell the interpreter on which track the config blocks are located (in sector 0 and 1).

`CUSTOM_FONT`

Tell the interpreter that a custom font will be used.

`FGCOL=n`
`FGCOLDM=n`

Set the foreground colour for normal mode and darkmode.

`NOSECTORPRELOAD`

Don't load any parts of static memory into RAM from disk sectors on boot. If make.rb can decide that this won't be needed anyway, it will set this flag, to make the interpreter a little smaller.

`SLOW`

Prioritise small size over speed, making the interpreter smaller and slower. For C128 and Plus/4, this is enabled by default and can't be disabled, due to their memory models.

`SOUND (MEGA65 only)`

Include support for sound (.wav sample files playback).

`STACK_PAGES=n`

Set the number of memory pages to use for stack.

`STATCOL=n`

Set the statusline colour. (only for z1-z3).

`TARGET_C128`
`TARGET_C64`
`TARGET_PLUS4`
`TARGET_MEGA65`

Set the target platform.

`TERPNO=n`

Set the interpreter number (0-19) as reported by the interpreter to the game. Default is 2 (Apple II) for Beyond Zork and 8 (C64) for other games.

`TRACE`

Allocate a page of memory to keep a record of the last 64 instructions that were executed. If both TRACE and DEBUG are defined, and the game crashes with a fatal error, the game prints out the ten last instructions that were executed (Z-machine address and opcode).

`UNSAFE`

Remove some checks for runtime errors, making the interpreter smaller and faster.

`VMEM`

Utilize virtual memory. Without this, the complete game must fit in C64 RAM available above the interpreter, all in all about 50 KB. Also check section "Virtual memory flags" below.

`Z1`
`Z2`
`Z3`
`Z4`
`Z5`
`Z7`
`Z8`

Build the interpreter to run Z-machine version 1, 2, 3, 4, 5, 7 or 8.

`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.

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_FLOPPY`

Print trace information for floppy operations.

`TRACE_FLOPPY_VERBOSE`

Print verbose trace information for floppy operations.

`TRACE_PRINT_ARRAYS`

Print the contents of the input array and the parse array whenever a read command has been performed.

`TRACE_READTEXT`

Print trace information for opcode read/aread/sread.

`TRACE_SHOW_DICT_ENTRIES`

Print which dictionary words are checked when trying to match a word, and if the input word was less than or greater than that word.

`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.

Configuration blocks

Interpreter configuration is stored in two blocks on the boot disk (track 1, 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 ---
2 bytes: Number of bytes used for vmem information, including this byte (high endian)
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:

z1/z2/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

1571 drive support:

- z1/z2/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.

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).

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.

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.