The Zork I Z-machine Interpreter

Robert Baruch

Contents

Chapter 1

Zork I

1.1 Introduction

Zork I: The Great Underground Empire was an Infocom text adventure originally written as part of Zork in 1977 by Tim Anderson, Marc Blank, Bruce Daniels, and Dave Lebling. The game runs under a virtual machine called the Z-Machine. Thus, only the Z-Machine interpreter needed to be ported for the game to be playable on various machines.

The purpose of this document is to reverse engineer the Z-Machine interpreter found in various versions of Zork I for the Apple II. The disk images used are from the Internet Archive:

- Zork I, revision 15 (`ZorkI_r15_4amCrack`)

The original Infocom assembly language files are available. The directory for the Apple II contains the original source code for various Z-Machine interpreters. Version 3 is called `ZIP`, version 4 is `EZIP`, version 5 is `XZIP`, and version 6 is `YZIP`. There is also a directory `OLDZIP` which seems to correspond to this version, version 2, although there are a few differences.

1.2 About this document

This is a literate programming document. This means the explanatory text is interspersed with source code. The source code can be extracted from the document and compiled.

The goal is to provide all the source code necessary to reproduce a binary identical to the one found on the Internet Archive's `ZorkI_r15_4amCrack` disk image.

The assembly code is assembled using `dasm`.

This document doesn't explain every last detail. It's assumed that the reader can find enough details on the 6502 processor and the Apple II series of computers to fill in the gaps.

Chapter 2

Programming techniques

2.1 Zero page temporaries

Zero-page consists essentially of global variables. Sometimes we need local temporaries, and Apple II programs mostly doesn't use the stack for those. Rather, some "global" variables are reserved for temporaries. You might see multiple symbols equated to a single zero-page location. The names of such symbols are used to make sense within their context.

2.2 Tail calls

Rather than a `JSR` immediately followed by an `RTS`, instead a `JMP` can be used to save stack space, code space, and time. This is known as a tail call, because it is a call that happens at the tail of a function.

2.3 Unconditional branches

The 6502 doesn't have an unconditional short jump. However, if you can find a condition that is always true, this can serve as an unconditional short jump, which saves space and time.

2.4 Stretchy branches

6502 branches have a limit to how far they can jump. If they really need to jump farther than that, you have to put a `JMP` or an unconditional branch within reach.

2.5 Shared code

To save space, sometimes code at the end of one function is also useful to the next function, as long as it is within reach. This can save space, at the expense of functions being completely independent.

2.6 Macros

The original Infocom source code uses macros for moving data around, and we will adopt these macros (with different names) and more to make our assembly language listings a little less verbose.

2.6.1 STOW, STOW2

`STOW` stores a 16-bit literal value to a memory location.

For example, `STOW #$01FF, $0200` stores the 16-bit value `#$01FF` to memory location `$0200` (of course in little-endian order).

This is the same as `MOVEI` in the original Infocom source code.

10 ⟨*Macros* 10⟩≡ (202a) 11a ▷

```
    MACRO STOW
        LDA      #<{1}
        STA      {2}
        LDA      #>{1}
        STA      {2}+1
    ENDM
```

Defines:
`STOW`, used in chunks 27–29, 54, 97, 100, 103, 107, 108, 113b, 115b, 117c, 119, 125, 130b, 139a, 143, 145, 147–51, 153b, 154a, and 201.

STOW2 does the same, but in the opposite order. Parts of the code were written by different programmers at different times, so it's possible that the MOVEI macro was used inconsistently.

11a ⟨*Macros* 10⟩+≡ (202a) ◁10 11b▷

```
    MACRO STOW2
        LDA     #>{1}
        STA     {2}+1
        LDA     #<{1}
        STA     {2}
    ENDM
```

Defines:
STOW2, used in chunk 108.

2.6.2 MOVB, MOVW, STOB

MOVB moves a byte from one memory location to another, while STOB stores a literal byte to a memory location. The implementation is identical, and the only difference is documentation.

For example, `MOVB $01, $0200` moves the byte at memory location `$01` to memory location `$0200`, while `STOB #$01, $0200` stores the byte `#$01` to memory location `$0200`.

These macros are the same as MOVE in the original Infocom source code.

11b ⟨*Macros* 10⟩+≡ (202a) ◁11a 12a▷

```
    MACRO MOVB
        LDA   {1}
        STA   {2}
    ENDM
    MACRO STOB
        LDA   {1}
        STA   {2}
    ENDM
```

Defines:
MOVB, used in chunks 34, 84a, 126a, 128–30, and 181a.
STOB, used in chunks 27, 28b, 62, 103, 112, 114d, 117a, 126a, 128a, and 131.

MOVW moves a 16-bit value from one memory location to the another.

For example, `MOVW $01FF, $A000` moves the 16-bit value at memory location `$01FF` to memory location `$A000`.

This is the same as MOVEW in the original Infocom source code.

12a ⟨*Macros* 10⟩+≡ (202a) ◁11b 12b▷

```
    MACRO MOVW
        LDA     {1}
        STA     {2}
        LDA     {1}+1
        STA     {2}+1
    ENDM
```

Defines:
MOVW, used in chunks 29b, 97, 100, 112, 114d, 116, 126a, 128–31, 137, 151b, 154b, 166b, 169b, 171–74, 176b, 177a, 179a, 180a, 182a, 183a, 185, 186, and 193.

2.6.3 PSHW, PULB, PULW

PSHW is a macro that pushes a 16-bit value in memory onto the 6502 stack.

For example, `PSHW $01FF` pushes the 16-bit value at memory location `$01FF` onto the 6502 stack.

This is the same as PUSHW in the original Infocom source code.

12b ⟨*Macros* 10⟩+≡ (202a) ◁12a 12c▷

```
    MACRO PSHW
        LDA     {1}
        PHA
        LDA     {1}+1
        PHA
    ENDM
```

Defines:
PSHW, used in chunks 97, 100, 160a, and 197.

PULB is a macro that pulls an 8-bit value from the 6502 stack to memory.

For example, `PULB $01FF` pulls an 8-bit value from the 6502 stack and stores it at memory location `$01FF`.

12c ⟨*Macros* 10⟩+≡ (202a) ◁12b 13a▷

```
    MACRO PULB
        PLA
        STA     {1}
    ENDM
```

Defines:
PULB, used in chunk 128b.

PULW is a macro that pulls a 16-bit value from the 6502 stack to memory.

For example, `PULW $01FF` pulls a 16-bit value from the 6502 stack and stores it at memory location `$01FF`.

This is the same as PULLW in the original Infocom source code.

13a ⟨*Macros* 10⟩+≡ (202a) ◁12c 13b▷

```
    MACRO PULW
        PLA
        STA     {1}+1
        PLA
        STA     {1}
    ENDM
```

Defines:
PULW, used in chunks 97, 100, 160a, and 197.

2.6.4 INCW

INCW is a macro that increments a 16-bit value in memory.

For example, `INCW $01FF` increments the 16-bit value at memory location `$01FF`.

This is the same as INCW in the original Infocom source code.

13b ⟨*Macros* 10⟩+≡ (202a) ◁13a 14a▷

```
    MACRO INCW
        INC     {1}
        BNE     .continue
        INC     {1}+1
.continue
    ENDM
```

Defines:
INCW, used in chunks 36, 107, 108, 137, 138, 160a, 174a, and 194.

2.6.5 ADDA, ADDAC, ADDB, ADDB2, ADDW, ADDWC

ADDA is a macro that adds the A register to a 16-bit memory location.

For example, ADDA $01FF adds the contents of the A register to the 16-bit value at memory location $01FF.

14a ⟨*Macros* 10⟩+≡ (202a) ◁13b 14b▷

```
        MACRO ADDA
            CLC
            ADC    {1}
            STA    {1}
            BCC    .continue
            INC    {1}+1
    .continue
        ENDM
```

Defines:
ADDA, used in chunks 91 and 130b.

ADDAC is a macro that adds the A register, and whatever the carry flag is set to, to a 16-bit memory location.

14b ⟨*Macros* 10⟩+≡ (202a) ◁14a 15a▷

```
        MACRO ADDAC
            ADC    {1}
            STA    {1}
            BCC    .continue
            INC    {1}+1
    .continue
        ENDM
```

Defines:
ADDAC, used in chunk 194.

ADDB is a macro that adds an 8-bit immediate value, or the 8-bit contents of memory, to a 16-bit memory location.

For example, `ADDB $01FF, #$01` adds the immediate value `#$01` to the 16-bit value at memory location `$01FF`, while `ADDB $01FF, $0300` adds the 8-bit value at memory location `$0300` to the 16-bit value at memory location `$01FF`.

This is the same as ADDB in the original Infocom source code. The immediate value is the second argument.

15a ⟨*Macros* 10⟩+≡ (202a) ◁14b 15b▷

```
    MACRO ADDB
        LDA     {1}
        CLC
        ADC     {2}
        STA     {1}
        BCC     .continue
        INC     {1}+1
.continue
    ENDM
```

Defines:
ADDB, used in chunks 145 and 147b.

ADDB2 is the same as ADDB except that it swaps the initial CLC and LDA instructions.

15b ⟨*Macros* 10⟩+≡ (202a) ◁15a 15c▷

```
    MACRO ADDB2
        CLC
        LDA     {1}
        ADC     {2}
        STA     {1}
        BCC     .continue
        INC     {1}+1
.continue
    ENDM
```

Defines:
ADDB2, used in chunks 92 and 93.

ADDW is a macro that adds two 16-bit values in memory and stores it to a third 16-bit memory location.

15c ⟨*Macros* 10⟩+≡ (202a) ◁15b 16a▷

```
    MACRO ADDW
        CLC
        ADDWC   {1}, {2}, {3}
    ENDM
```

Defines:
ADDW, used in chunks 73, 90, 140, 165–68, and 170b.
Uses ADDWC 16a.

ADDWC is a macro that adds two 16-bit values in memory, plus the carry bit, and stores it to a third 16-bit memory location.

16a ⟨*Macros* 10⟩+≡ (202a) ◁15c 16b▷

```
    MACRO ADDWC
        LDA    {1}
        ADC    {2}
        STA    {3}
        LDA    {1}+1
        ADC    {2}+1
        STA    {3}+1
    ENDM
```

Defines:
ADDWC, used in chunks 15c and 97.

2.6.6 SUBB, SUBB2, SUBW

SUBB is a macro that subtracts an 8-bit value from a 16-bit memory location. This is the same as SUBB in the original Infocom source code. The immediate value is the second argument.

16b ⟨*Macros* 10⟩+≡ (202a) ◁16a 17a▷

```
    MACRO SUBB
        LDA    {1}
        SEC
        SBC    {2}
        STA    {1}
        BCS    .continue
        DEC    {1}+1
.continue
    ENDM
```

Defines:
SUBB, used in chunks 35, 93, 130c, 160b, 163b, and 182a.

SUBB2 is the same as SUBB except that it swaps the initial SEC and LDA instructions.

17a ⟨*Macros* 10⟩+≡ (202a) ◁16b 17b▷
```
    MACRO SUBB2
        SEC
        LDA     {1}
        SBC     {2}
        STA     {1}
        BCS     .continue
        DEC     {1}+1
.continue
    ENDM
```
Defines:
SUBB2, used in chunk 92b.

SUBW is a macro that subtracts the 16-bit memory value in the second argument from a 16-bit memory location in the first argument, and stores it in the 16-bit memory location in the third argument.

17b ⟨*Macros* 10⟩+≡ (202a) ◁17a 17c▷
```
    MACRO SUBW
        SEC
        LDA     {1}
        SBC     {2}
        STA     {3}
        LDA     {1}+1
        SBC     {2}+1
        STA     {3}+1
    ENDM
```
Defines:
SUBW, used in chunks 94b, 95a, 174c, and 194.

2.6.7 ROLW, RORW

ROLW rotates a 16-bit memory location left.

17c ⟨*Macros* 10⟩+≡ (202a) ◁17b 18▷
```
    MACRO ROLW
        ROL     {1}
        ROL     {1}+1
    ENDM
```
Defines:
ROLW, used in chunk 100.

RORW rotates a 16-bit memory location right.

18 ⟨*Macros* 10⟩+≡ (202a) ◁17c

```
MACRO RORW
    ROR    {1}+1
    ROR    {1}
ENDM
```

Defines:
RORW, used in chunk 97.

Chapter 3

The boot process

Suggested reading: *Beneath Apple DOS* (Don Worth, Pieter Lechner, 1982) page 5-6, "What happens during booting".

We will only examine the boot process in order to get to the main program. The boot process may just be the way the 4am disk image works, so should not be taken as original to Zork.

We will be doing a deep dive into `BOOT1`, since it is fairly easy to understand.

Apple II programs originally came on disk, and such disks are generally bootable. You'd put the disk in Drive 1, reset the computer, and the disk card ROM then loads the `BOOT1` section of the disk. This section starts from track 0 sector 0, and is almost always 1 sector (256 bytes) long. The data is stored to location `$0800` and then the disk card ROM causes the CPU to jump to location `$0801`. The very first byte in track 0 sector 0 is the number of sectors in this `BOOT1` section, and again, this is almost always `1`.

After the disk card reads `BOOT1`, the zero-page location `IWMDATAPTR` is left as the pointer to the buffer to next read data into, so `$0900`. The location `IWMSLTNDX` is the disk card's slot index (slot times 16).

3.1 BOOT1

`BOOT1` reads a number of sectors from track 0, backwards from a starting sector, down to sector 0. The sector to read is stored in `BOOT1_SECTOR_NUM`, and is initially `9` for Zork I release 15. The RAM address to read the sectors to is stored in BOOT1_WRITE_ADDR, and it is $2200. Thus, BOOT1 will read sectors 0 through 9 into address $2200 - $2BFF.

20a ⟨*BOOT1* 20a⟩≡ 22d ▷

```
        BYTE    #$01   ; Number of sectors in BOOT1. Almost always 1.
    BOOT1:
        SUBROUTINE

    ⟨Read BOOT2 from disk 20c⟩
    ⟨Jump to BOOT2 25⟩
    ⟨BOOT1 parameters 20b⟩
```

Defines:
BOOT1, used in chunk 24.

20b ⟨*BOOT1 parameters* 20b⟩≡ (20a)

```
        ORG     $08FD

    BOOT1_WRITE_ADDR:
        HEX     00 22
    BOOT1_SECTOR_NUM:
        HEX     09
```

Defines:
BOOT1_SECTOR_NUM, used in chunks 21, 22, and 24.
BOOT1_WRITE_ADDR, used in chunks 21–24.

Reading BOOT2 involves repeatedly calling the disk card ROM's sector read routine with appropriate parameters. But first, we have to initialize some variables.

20c ⟨*Read BOOT2 from disk* 20c⟩≡ (20a) 23 ▷

```
    ⟨Skip initialization if BOOT1 already initialized 21a⟩
    .init_vars:
    ⟨Initialize BOOT1 21b⟩

    .already_initted:
    ⟨Set up parameters for reading a sector 22a⟩
        JMP     (RDSECT_PTR)
```

Uses RDSECT_PTR 203.

The reason we have to check whether BOOT1 has already been initialized is that the disk card ROM's RDSECT routine jumps back to BOOT1 after reading a sector.

Checking for initialization is as simple as checking the IWMDATAPTR page against 09. If it's 09 then we have just finished reading BOOT1, and this is the first call to BOOT1, so we need to initialize. Otherwise, we can skip initialization.

21a ⟨*Skip initialization if BOOT1 already initialized* 21a⟩≡ (20c)

```
    LDA     IWMDATAPTR+1
    CMP     #$09
    BNE     .already_initted
```

Uses IWMDATAPTR 203.

To initialize the BOOT1 variables, we first determine the disk card ROM's RDSECT routine address. This is simply $CX5C, where X is the disk card's slot number.

21b ⟨*Initialize BOOT1* 21b⟩≡ (20c) 21c ▷

```
    LDA     IWMSLTNDX
    LSR
    LSR
    LSR
    LSR
    ORA     #$C0
    STA     RDSECT_PTR+1
    LDA     #$5C
    STA     RDSECT_PTR
```

Uses IWMSLTNDX 203 and RDSECT_PTR 203.

Next, we initialize the address to read disk data into. Since we're reading backwards, we start by adding BOOT1_SECTOR_NUM to the page number in BOOT1_WRITE_ADDR.

21c ⟨*Initialize BOOT1* 21b⟩+≡ (20c) ◁ 21b

```
    CLC
    LDA     BOOT1_WRITE_ADDR+1
    ADC     BOOT1_SECTOR_NUM
    STA     BOOT1_WRITE_ADDR+1
```

Uses BOOT1_SECTOR_NUM 20b and BOOT1_WRITE_ADDR 20b.

Now that BOOT1 has been initialized, we can set up the parameters for the next read. This means loading up IWMSECTOR with the sector in track 0 to read, IWMDATAPTR with the address to read data into, and loading the X register with the slot index (slot times 16).

First we check whether we've read all sectors by checking whether BOOT1_SECTOR_NUM is less than zero - recall that we are reading sectors from last down to 0.

22a ⟨*Set up parameters for reading a sector* 22a⟩≡ (20c) 22c▷

```
      LDX     BOOT1_SECTOR_NUM
      BMI     .go_to_boot2      ; Are we done?
```

Uses BOOT1_SECTOR_NUM 20b.

We set up IWMSECTOR by taking the sector number and translating it to a physical sector on the disk using a translation table. This has to do with the way sectors on disk are interleaved for efficiency.

22b ⟨*BOOT1 sector translation table* 22b⟩≡ (22d)

```
      ORG     $084D

  BOOT1_SECTOR_TRANSLATE_TABLE:
      HEX     00 0D 0B 09 07 05 03 01
      HEX     0E 0C 0A 08 06 04 02 0F
```

Defines:
BOOT1_SECTOR_TRANSLATE_TABLE, used in chunks 22c and 24.

22c ⟨*Set up parameters for reading a sector* 22a⟩+≡ (20c) ◁22a 22e▷

```
      LDA     BOOT1_SECTOR_TRANSLATE_TABLE,X
      STA     IWMSECTOR
```

Uses BOOT1_SECTOR_TRANSLATE_TABLE 22b and IWMSECTOR 203.

22d ⟨*BOOT1* 20a⟩+≡ ◁20a 24▷
⟨*BOOT1 sector translation table* 22b⟩

Then we transfer BOOT1_WRITE_ADDR into IWMDATAPTR, decrement BOOT1_SECTOR_NUM, and load up the X register with IWMSLTNDX.

22e ⟨*Set up parameters for reading a sector* 22a⟩+≡ (20c) ◁22c

```
      DEC     BOOT1_SECTOR_NUM
      LDA     BOOT1_WRITE_ADDR+1
      STA     IWMDATAPTR+1
      DEC     BOOT1_WRITE_ADDR+1
      LDX     IWMSLTNDX
```

Uses BOOT1_SECTOR_NUM 20b, BOOT1_WRITE_ADDR 20b, IWMDATAPTR 203, and IWMSLTNDX 203.

Once BOOT1 has finished loading, it jumps to the second page it loaded, which is from sector 1. This is called BOOT2.

23 ⟨*Read BOOT2 from disk* 20c⟩+≡ (20a) ◁20c

```
  .go_to_boot2
      INC      BOOT1_WRITE_ADDR+1
      INC      BOOT1_WRITE_ADDR+1

      ; Set keyboard and screen as I/O, set all soft switches to defaults,
      ; e.g. text mode, lores graphics, etc.

      JSR      SETKBD
      JSR      SETVID
      JSR      INIT

      ; Go to BOOT2!

      LDX      IWMSLTNDX
      JMP      (BOOT1_WRITE_ADDR)
```

Uses BOOT1_WRITE_ADDR 20b, INIT 203, IWMSLTNDX 203, SETKBD 203, and SETVID 203.

24 ⟨*BOOT1* 20a⟩+≡ ◁22d

```
    ; Initially, IWMDATAPTR is left with 0900 by the disk card. We initialize
    ; some of our vars only once, so we check IWMDATAPTR+1 to see if it's
    ; 09. If it is, we haven't yet initialized.

    LDA     IWMDATAPTR+1
    CMP     #$09
    BNE     .already_initted

.init_vars:
    ; Set the RDSECT_PTR to $CX5C, where X is the slot number
    ; of the disk card.

    LDA     IWMSLTNDX
    LSR
    LSR
    LSR
    LSR
    ORA     #$C0
    STA     RDSECT_PTR+1
    LDA     #$5C
    STA     RDSECT_PTR

    ; Add BOOT1_SECTOR_NUM to the BOOT1_WRITE_ADDR page, since we will read
    ; backwards from BOOT1_SECTOR_NUM.

    CLC
    LDA     BOOT1_WRITE_ADDR+1
    ADC     BOOT1_SECTOR_NUM
    STA     BOOT1_WRITE_ADDR+1

.already_initted:
    LDX     BOOT1_SECTOR_NUM
    BMI     .go_to_boot2        ; Are we done?

    ; Translate logical sector to physical sector. This has to do with the way
    ; sectors on disk are interleaved for efficiency.

    LDA     BOOT1_SECTOR_TRANSLATE_TABLE,X
    STA     IWMSECTOR
    DEC     BOOT1_SECTOR_NUM
    LDA     BOOT1_WRITE_ADDR+1
    STA     IWMDATAPTR+1
    DEC     BOOT1_WRITE_ADDR+1
    LDX     IWMSLTNDX

    ; The disk card's read sector function jumps back to BOOT1 after reading the
    ; sector. The sector to read is in IWMSECTOR, and the page to write
    ; the data to is in IWMDATAPTR+1. The X register contains the disk slot
    ; times 16.
```

```
    JMP     (RDSECT_PTR)

.go_to_boot2
    ; BOOT2 starts with sector 1, not sector 0, so increment the page from
    ; BOOT1_WRITE_ADDR by 2.

    INC     BOOT1_WRITE_ADDR+1
    INC     BOOT1_WRITE_ADDR+1

    ; Set keyboard and screen as I/O, set all soft switches to defaults,
    ; e.g. text mode, lores graphics, etc.

    JSR     SETKBD
    JSR     SETVID
    JSR     INIT

    ; Go to BOOT2!

    LDX     IWMSLTNDX
    JMP     (BOOT1_WRITE_ADDR)

    ORG     $084D
BOOT1_SECTOR_TRANSLATE_TABLE:
    HEX     00 0D 0B 09 07 05 03 01
    HEX     0E 0C 0A 08 06 04 02 0F
```

Uses BOOT1 20a, BOOT1_SECTOR_NUM 20b, BOOT1_SECTOR_TRANSLATE_TABLE 22b, BOOT1_WRITE_ADDR 20b, INIT 203, IWMDATAPTR 203, IWMSECTOR 203, IWMSLTNDX 203, RDSECT_PTR 203, SETKBD 203, and SETVID 203.

25 ⟨*Jump to BOOT2* 25⟩≡ (20a)

3.2 BOOT2

BOOT2 loads 26 sectors starting from track 1 sector 0 into addresses $0800-$21FF, and then jumps to $0800. Normally, BOOT2 loads DOS and jumps to it, but in this case we don't need DOS and go directly to the main program.

Chapter 4

The main program

This is the Z-machine proper.

We first clear out the top half of zero page ($80-$FF).

26a ⟨*main* 26a⟩≡ (207) 26b ▷

```
main:
    SUBROUTINE

    CLD
    LDA     #$00
    LDX     #$80

.clear:
    STA     $80,X
    INX
    BNE     .clear
```

Defines:
main, used in chunks 29b, 30, 40, 43, and 200b.

And we reset the 6502 stack pointer.

26b ⟨*main* 26a⟩+≡ (207) ◁ 26a 27 ▷

```
    LDX     #$FF
    TXS
```

Next, we set up some variables. The printer output routine, `PRINTER_CSW`, is set to `$C100`. This is the address of the ROM of the card in slot 1, which is typically the printer card. It will be used later when outputting text to both screen and printer.

Next, we set `ZCODE_PAGE_VALID` to zero, which will later cause the Z-machine to load the first page of Z-code into memory when the first instruction is retrieved.

The z-stack count, `STACK_COUNT`, is set to 1, and the z-stack pointer, `Z_SP`, is set to `$03E8`.

There are two page tables, `PAGE_L_TABLE` and `PAGE_H_TABLE`, which are set to `$2200` and `$2280`, respectively. These are used to map Z-machine memory pages to physical memory pages.

There are two other page tables, `NEXT_PAGE_TABLE` and `PREV_PAGE_TABLE`, which are set to `$2300` and `$2380`, respectively. Together this forms a doubly-linked list of pages.

27 ⟨*main* 26a⟩+≡ (207) ◁26b 28a▷

```
.set_vars:
    ; Historical note: Setting PRINTER_CSW was originally a call to SINIT,
    ; "system-dependent initialization".
    LDA     #$C1
    STA     PRINTER_CSW+1
    LDA     #$00
    STA     PRINTER_CSW
    LDA     #$00
    STA     ZCODE_PAGE_VALID
    STA     ZCODE_PAGE_VALID2
    STOB    #$01, STACK_COUNT
    STOW    #$03E8, Z_SP
    STOB    #$FF, ZCHAR_SCRATCH1+6
    STOW    #$2200, PAGE_L_TABLE
    STOW    #$2280, PAGE_H_TABLE
    STOW    #$2300, NEXT_PAGE_TABLE
    STOW    #$2380, PREV_PAGE_TABLE
```

Uses `NEXT_PAGE_TABLE` 204, `PAGE_H_TABLE` 204, `PAGE_L_TABLE` 204, `PREV_PAGE_TABLE` 204, `PRINTER_CSW` 204, `STACK_COUNT` 204, `STOB` 11b, `STOW` 10, `ZCHAR_SCRATCH1` 204, `ZCODE_PAGE_VALID` 204, `ZCODE_PAGE_VALID2` 204, and `Z_SP` 204.

Next, we initialize the page tables. This zeros out PAGE_L_TABLE and PAGE_H_TABLE, and then sets up the next and previous page tables. NEXT_PAGE_TABLE is initialized to `01 02 03 ... 7F FF` and so on, while PREV_PAGE_TABLE is initialized to `FF 00 01 ... 7D 7E`. `FF` is the null pointer for this linked list.

28a ⟨*main* 26a⟩+≡ (207) ◁27 28b▷

```
    LDY     #$00
    LDX     #$80        ; Max pages

.loop_inc_dec_tables:
    LDA     #$00
    STA     (PAGE_L_TABLE),Y
    STA     (PAGE_H_TABLE),Y
    TYA
    CLC
    ADC     #$01
    STA     (NEXT_PAGE_TABLE),Y
    TYA
    SEC
    SBC     #$01
    STA     (PREV_PAGE_TABLE),Y
    INY
    DEX
    BNE     .loop_inc_dec_tables
    DEY
    LDA     #$FF
    STA     (NEXT_PAGE_TABLE),Y
```

Uses NEXT_PAGE_TABLE 204, PAGE_H_TABLE 204, PAGE_L_TABLE 204, and PREV_PAGE_TABLE 204.

Next, we set FIRST_Z_PAGE to 0 (the head of the list), LAST_Z_PAGE to #$7F (the tail of the list), and Z_HEADER_ADDR to $2C00. Z_HEADER_ADDR is the address in memory where the Z-code image header is stored.

28b ⟨*main* 26a⟩+≡ (207) ◁28a 28c▷

```
    STOB    #$00, FIRST_Z_PAGE
    STOB    #$7F, LAST_Z_PAGE
    STOW    #$2C00, Z_HEADER_ADDR
```

Uses FIRST_Z_PAGE 204, LAST_Z_PAGE 204, STOB 11b, and STOW 10.

Then we clear the screen.

28c ⟨*main* 26a⟩+≡ (207) ◁28b 29b▷

```
    JSR     do_reset_window
```

Uses do_reset_window 29a.

29a ⟨*Do reset window* 29a⟩≡ (207)
```
  do_reset_window:
      JSR      reset_window
      RTS
```
Defines:
  `do_reset_window`, used in chunk 28c.
Uses `reset_window` 49.

Next, we start reading the image of Z-code from disk into memory. The first page of the image, which is the image header, gets loaded into the address stored in `Z_HEADER_ADDR`. This done through the `read_from_sector` routine, which reads the (256 byte) sector stored in `SCRATCH1`, relative to track 3 sector 0, into the address stored in `SCRATCH2`.

If there was an error reading, we jump back to the beginning of the main program and start again. This would result in a failure loop with no apparent output if the disk is damaged.

29b ⟨*main* 26a⟩+≡ (207) ◁28c 30▷
```
  .read_z_image:
      MOVW     Z_HEADER_ADDR, SCRATCH2
      STOW     #$0000, SCRATCH1
      JSR      read_from_sector

      ; Historical note: The original Infocom source code did not check
      ; for an error here.

      BCC      .no_error
      JMP      main
```
Uses `MOVW` 12a, `SCRATCH1` 204, `SCRATCH2` 204, `STOW` 10, `main` 26a, and `read_from_sector` 107.

If there was no error reading the image header, we write `#$FF` into byte 5 of the header, whose purpose is not known at this point. Then we load byte 4 of the header, which is the page for the "base of high memory", and store it (plus 1) in `NUM_IMAGE_PAGES`.

Then, we read `NUM_IMAGE_PAGES-1` consecutive sectors after the header into consecutive memory.

Suppose `Z_HEADER_ADDR` is `$2C00`. We have already read the header sector in. Now suppose the base of high memory in the header is `#$01F6`. Then `NUM_IMAGE_PAGES` would be `#$02`, and we would read one sector into memory at `$2D00`.

In the case of Zork I, `Z_HEADER_ADDR` is `$2C00`, and the base of high memory is `#$47FF`. `NUM_IMAGE_PAGES` is thus `#$48`. So, we would read 71 more sectors into memory, from `$2D00` to `$73FF`.

30 ⟨*main* 26a⟩+≡ (207) ◁29b 31a▷

```
.no_error:
    LDY     #$05
    LDA     #$FF
    STA     (Z_HEADER_ADDR),Y
    DEY
    LDA     (Z_HEADER_ADDR),Y
    STA     NUM_IMAGE_PAGES
    INC     NUM_IMAGE_PAGES
    LDA     #$00

.read_another_sector:
    CLC                         ; "START2"
    ADC     #$01
    TAX
    ADC     Z_HEADER_ADDR+1
    STA     SCRATCH2+1
    LDA     Z_HEADER_ADDR
    STA     SCRATCH2
    TXA
    CMP     NUM_IMAGE_PAGES
    BEQ     .check_bit_0_flag     ; done loading
    PHA
    STA     SCRATCH1
    LDA     #$00
    STA     SCRATCH1+1
    JSR     read_from_sector

    ; Historical note: The original Infocom source code did not check
    ; for an error here.

    BCC     .no_error2
    JMP     main
```

```
  .no_error2:
      PLA
      JMP      .read_another_sector
```
Uses NUM_IMAGE_PAGES 204, SCRATCH1 204, SCRATCH2 204, main 26a, and read_from_sector 107.

Next, we check the debug-on-start flag stored in bit 0 of byte 1 of the header, and if it isn't clear, we execute a BRK instruction. That drops the Apple II into its monitor, which allows debugging, however primitive by our modern standards.

This part was not in the original Infocom source code.

31a ⟨*main* 26a⟩+≡ (207) ◁30 31d▷
```
  .check_bit_0_flag:
      LDY      #$01
      LDA      (Z_HEADER_ADDR),Y
      AND      #$01
      EOR      #$01
      BEQ      .brk
```
Uses brk 31c.

31b ⟨*die* 31b⟩≡ (33b)
```
  .brk:
      JSR      brk
```
Uses brk 31c.

31c ⟨*brk* 31c⟩≡ (207)
```
  brk:
      BRK
```
Defines:
brk, used in chunks 31, 33a, 35, 36, 158, 177b, 193, and 198.

Continuing after the load, we set the 24-bit Z_PC program counter to its initial 16-bit value, which is stored in the header at bytes 6 and 7, bigendian. For Zork I, Z_PC becomes #$004859.

31d ⟨*main* 26a⟩+≡ (207) ◁31a 32▷
```
  .store_initial_z_pc:
      LDY      #$07
      LDA      (Z_HEADER_ADDR),Y
      STA      Z_PC
      DEY
      LDA      (Z_HEADER_ADDR),Y
      STA      Z_PC+1
      LDA      #$00
      STA      Z_PC+2
```
Uses Z_PC 204.

Next, we load GLOBAL_ZVARS_ADDR and Z_ABBREV_TABLE from the header at bytes #$0C-$0D and #$18-$19, respectively. Again, these are bigendian values, so get byte-swapped. These are relative to the beginning of the image, so we simply add the page of the image address to them. There is no need to add the low byte of the header address, since the header already begins on a page boundary.

For Zork I, the header values are #$20DE and #$00CA, respectively. This means that GLOBAL_ZVARS_ADDR is $4CDE and Z_ABBREV_TABLE is $2CCA.

32 ⟨*main* 26a⟩+≡ (207) ◁31d 33a▷

```
.store_z_global_vars_addr:
    LDY     #$0D
    LDA     (Z_HEADER_ADDR),Y
    STA     GLOBAL_ZVARS_ADDR
    DEY
    LDA     (Z_HEADER_ADDR),Y
    CLC
    ADC     Z_HEADER_ADDR+1
    STA     GLOBAL_ZVARS_ADDR+1

.store_z_abbrev_table_addr:
    LDY     #$19
    LDA     (Z_HEADER_ADDR),Y
    STA     Z_ABBREV_TABLE
    DEY
    LDA     (Z_HEADER_ADDR),Y
    CLC
    ADC     Z_HEADER_ADDR+1
    STA     Z_ABBREV_TABLE+1
```

Uses GLOBAL_ZVARS_ADDR 204 and Z_ABBREV_TABLE 204.

Next, we set AFTER_Z_IMAGE_ADDR to the page-aligned memory address immediately after the image, and compare its page to the last viable RAM page. If it is greater, we hit a BRK instruction since there isn't enough memory to run the game.

For Zork I, AFTER_Z_IMAGE_ADDR is $7400.

For a fully-populated Apple II (64k RAM), the last viable RAM page is #$BF.

33a ⟨*main* 26a⟩+≡ (207) ◁32 33b▷

```
    LDA     #$00
    STA     AFTER_Z_IMAGE_ADDR
    LDA     NUM_IMAGE_PAGES
    CLC
    ADC     Z_HEADER_ADDR+1
    STA     AFTER_Z_IMAGE_ADDR+1
    JSR     locate_last_ram_page
    SEC
    SBC     AFTER_Z_IMAGE_ADDR+1
    BCC     .brk
```

Uses AFTER_Z_IMAGE_ADDR 204, NUM_IMAGE_PAGES 204, and brk 31c.

We then store the difference as the last Z-image page in LAST_Z_PAGE, and the same, plus 1, in FIRST_Z_PAGE. We also set the next page table entry of the last page to #$FF.

For Zork I, FIRST_Z_PAGE is #$4C, and LAST_Z_PAGE is #$4B.

And lastly, we start the interpreter loop by executing the first instruction in z-code.

33b ⟨*main* 26a⟩+≡ (207) ◁33a

```
    TAY
    INY
    STY     FIRST_Z_PAGE
    TAY
    STY     LAST_Z_PAGE
    LDA     #$FF
    STA     (NEXT_PAGE_TABLE),Y
    JMP     do_instruction

  ⟨die 31b⟩
```

Uses FIRST_Z_PAGE 204, LAST_Z_PAGE 204, NEXT_PAGE_TABLE 204, and do_instruction 112.

To locate the last viable RAM page, we start with `$C0FF` in `SCRATCH2`.

We then decrement the high byte of `SCRATCH2`, and read from the address twice. If it reads differently, we are not yet into viable RAM, so we decrement and try again.

Otherwise, we invert the byte, write it back, and read it back. Again, if it reads differently, we decrement and try again.

Finally, we return the high byte of `SCRATCH2`.

34 ⟨*Locate last RAM page* 34⟩≡ (207)

```
locate_last_ram_page:
    SUBROUTINE

    MOVB    #$C0, SCRATCH2+1
    MOVB    #$FF, SCRATCH2
    LDY     #$00

.loop:
    DEC     SCRATCH2+1
    LDA     (SCRATCH2),Y
    CMP     (SCRATCH2),Y
    BNE     .loop
    EOR     #$FF
    STA     (SCRATCH2),Y
    CMP     (SCRATCH2),Y
    BNE     .loop
    EOR     #$FF
    STA     (SCRATCH2),Y
    LDA     SCRATCH2+1
    RTS
```

Defines:
`locate_last_ram_addr`, never used.
Uses `MOVB` 11b and `SCRATCH2` 204.

Chapter 5

The Z-stack

The Z-stack is a stack of 16-bit values used by the Z-machine. It is not the same as the 6502 stack. The stack can hold values, but also holds call frames (see Call). The stack grows downwards in memory.

The stack pointer is `Z_SP`, and it points to the current top of the stack. The counter `STACK_COUNT` contains the current number of 16-bit elements on the stack.

As mentioned above, `STACK_COUNT`, is initialized to 1 and `Z_SP`, is initialized to `$03E8`.

Pushing a 16-bit value onto the stack involves placing the value at the next two free locations, low byte first, and then decrementing the stack pointer by 2. So for example, if pushing the value `#$1234` onto the stack, and `Z_SP` is `$03E8`, then `$03E7` will contain `#$34`, `$03E6` will contain `#$12`, and `Z_SP` will end up as `$03E6`. `STACK_COUNT` will also be incremented.

The `push` routine pushes the 16-byte value in `SCRATCH2` onto the stack. According to the code, if the number of elements becomes `#$B4` (180), the program will hit a BRK instruction.

35 ⟨*Push* 35⟩≡ (207)

```
push:
    SUBROUTINE

    SUBB    Z_SP, #$01
    LDY     #$00
    LDA     SCRATCH2
    STA     (Z_SP),Y
    SUBB    Z_SP, #$01
    LDA     SCRATCH2+1
```

```
        STA      (Z_SP),Y
        INC      STACK_COUNT
        LDA      STACK_COUNT
        CMP      #$B4
        BCC      .end
        JSR      brk

    .end:
        RTS
```

Defines:
push, used in chunks 123, 124, 126–28, and 169b.
Uses SCRATCH2 204, STACK_COUNT 204, SUBB 16b, Z_SP 204, and brk 31c.

The pop routine pops a 16-bit value from the stack into SCRATCH2, which increments Z_SP by 2, then decrements STACK_COUNT. If STACK_COUNT ends up as zero, the stack underflows and the program will hit a BRK instruction.

36 ⟨*Pop* 36⟩≡ (207)

```
    pop:
        SUBROUTINE

        LDY      #$00
        LDA      (Z_SP),Y
        STA      SCRATCH2+1
        INCW     Z_SP
        LDA      (Z_SP),Y
        STA      SCRATCH2
        INCW     Z_SP
        DEC      STACK_COUNT
        BNE      .end
        JSR      brk
    .end:
        RTS
```

Defines:
pop, used in chunks 121, 124, 130, 168b, 169a, and 183a.
Uses INCW 13b, SCRATCH2 204, STACK_COUNT 204, Z_SP 204, and brk 31c.

Chapter 6

Z-code

Z-code is not stored in memory in a linear fashion. Rather, it is stored in pages of 256 bytes, in the order that the Z-machine loads them. `ZCODE_PAGE_ADDR` is the address in memory that the current page of Z-code is stored in.

The `Z_PC` 24-bit address is an address into z-code. So, getting the next code byte translates to retrieving the byte at `(ZCODE_PAGE_ADDR) + Z_PC` and incrementing the low byte of `Z_PC`.

Of course, if the low byte of `Z_PC` ends up as 0, we'll need to propagate the increment to its other bytes, but also invalidate the current code page.

This is handled through the `ZCODE_PAGE_VALID` flag. If it is zero, then we will need to load a page of Z-code into `ZCODE_PAGE_ADDR`.

As an example, when the Z-machine starts, `Z_PC` is `#$004859`, and `ZCODE_PAGE_VALID` is `0`. This means that we will have to load code page `#$48`.

37 ⟨*Get next code byte* 37⟩≡ (207) 38▷

```
  get_next_code_byte:
      SUBROUTINE

      LDA     ZCODE_PAGE_VALID
      BEQ     .zcode_page_invalid
      LDY     Z_PC                    ; load from memory
      LDA     (ZCODE_PAGE_ADDR),Y
      INY
      STY     Z_PC
      BEQ     .invalidate_zcode_page     ; next byte in next page?
      RTS

  .invalidate_zcode_page:
```

```
        LDY     #$00
        STY     ZCODE_PAGE_VALID
        INC     Z_PC+1
        BNE     .end
        INC     Z_PC+2

    .end:
        RTS
```

Defines:
`get_next_code_byte`, used in chunks 39, 112, 113a, 120, 121, 123, 126c, 127, 161, and 162.
Uses `ZCODE_PAGE_ADDR` 204, `ZCODE_PAGE_VALID` 204, and `Z_PC` 204.

As an example, on start, `Z_PC` is `#$004859`, so we have to access code page `#$0048`. Since the high byte isn't set, we know that the code page is in memory. If the high byte were set, we would have to locate that page in memory, and if it isn't there, we would have to load it from disk.

But let's suppose that `Z_PC` were `#$014859`. We would have to access code page `#$0148`. Initially, `PAGE_L_TABLE` and `PAGE_H_TABLE` are zeroed out, so `find_index_of_page_table` would return with carry set and the `A` register set to `LAST_Z_PAGE` (`#$4B`).

38 ⟨*Get next code byte* 37⟩+≡ (207) ◁37 39▷

```
    .zcode_page_invalid:
        LDA     Z_PC+2
        BNE     .find_pc_page_in_page_table
        LDA     Z_PC+1
        CMP     NUM_IMAGE_PAGES
        BCC     .set_page_addr

    .find_pc_page_in_page_table:
        LDA     Z_PC+1
        STA     SCRATCH2
        LDA     Z_PC+2
        STA     SCRATCH2+1
        JSR     find_index_of_page_table
        STA     PAGE_TABLE_INDEX
        BCS     .not_found_in_page_table

    .set_page_first:
        JSR     set_page_first
        CLC
        LDA     PAGE_TABLE_INDEX
        ADC     NUM_IMAGE_PAGES
```

Defines:
`.zcode_page_invalid`, never used.
Uses `NUM_IMAGE_PAGES` 204, `PAGE_TABLE_INDEX` 204, `SCRATCH2` 204, `Z_PC` 204, `find_index_of_page_table` 41, and `set_page_first` 42.

Once we've ensured that the desired Z-code page is in memory, we can add the page to the page of Z_HEADER_ADDR and store in ZCODE_PAGE_ADDR. We also set the low byte of ZCODE_PAGE_ADDR to zero since we're guaranteed to be at the top of the page. We also set ZCODE_PAGE_VALID to true. And finally we go back to the beginning of the routine to get the next code byte.

39 ⟨*Get next code byte* 37⟩+≡ (207) ◁38 40▷

```
.set_page_addr:
    CLC
    ADC         Z_HEADER_ADDR+1
    STA         ZCODE_PAGE_ADDR+1
    LDA         #$00
    STA         ZCODE_PAGE_ADDR
    LDA         #$FF
    STA         ZCODE_PAGE_VALID
    JMP         get_next_code_byte
```

Defines:
 .set_page_addr, never used.
Uses ZCODE_PAGE_ADDR 204, ZCODE_PAGE_VALID 204, and get_next_code_byte 37.

If the page we need isn't found in the page table, we need to load it from disk, and it gets loaded into AFTER_Z_IMAGE_ADDR plus PAGE_TABLE_INDEX pages. On a good read, we store the z-page value into the page table.

40 ⟨*Get next code byte* 37⟩+≡ (207) ◁39

```
.not_found_in_page_table:
    CMP     PAGE_TABLE_INDEX2
    BNE     .read_from_disk
    LDA     #$00
    STA     ZCODE_PAGE_VALID2

.read_from_disk:
    LDA     AFTER_Z_IMAGE_ADDR
    STA     SCRATCH2
    LDA     AFTER_Z_IMAGE_ADDR+1
    STA     SCRATCH2+1
    LDA     PAGE_TABLE_INDEX
    CLC
    ADC     SCRATCH2+1
    STA     SCRATCH2+1
    LDA     Z_PC+1
    STA     SCRATCH1
    LDA     Z_PC+2
    STA     SCRATCH1+1
    JSR     read_from_sector
    BCC     .good_read
    JMP     main

.good_read:
    LDY     PAGE_TABLE_INDEX
    LDA     Z_PC+1
    STA     (PAGE_L_TABLE),Y
    LDA     Z_PC+2
    STA     (PAGE_H_TABLE),Y
    TYA
    JMP     .set_page_first
```

Defines:
  .not_found_in_page_table, never used.

Uses AFTER_Z_IMAGE_ADDR 204, PAGE_H_TABLE 204, PAGE_L_TABLE 204, PAGE_TABLE_INDEX 204, PAGE_TABLE_INDEX2 204, SCRATCH1 204, SCRATCH2 204, ZCODE_PAGE_VALID2 204, Z_PC 204, main 26a, read_from_sector 107, and set_page_first 42.

Given a page-aligned address in SCRATCH2, this routine searches through the PAGE_L_TABLE and PAGE_H_TABLE for that address, returning the index found in A (or LAST_Z_PAGE if not found). The carry flag is clear if the page was found, otherwise it is set.

41 ⟨*Find index of page table* 41⟩≡ (207)

```
find_index_of_page_table:
    SUBROUTINE

    LDX      FIRST_Z_PAGE
    LDY      #$00
    LDA      SCRATCH2

.loop:
    CMP      (PAGE_L_TABLE),Y
    BNE      .next
    LDA      SCRATCH2+1
    CMP      (PAGE_H_TABLE),Y
    BEQ      .found
    LDA      SCRATCH2

.next:
    INY
    DEX
    BNE      .loop
    LDA      LAST_Z_PAGE
    SEC
    RTS

.found:
    TYA
    CLC
    RTS
```

Defines:
find_index_of_page_table, used in chunks 38 and 43.

Uses FIRST_Z_PAGE 204, LAST_Z_PAGE 204, PAGE_H_TABLE 204, PAGE_L_TABLE 204, and SCRATCH2 204.

Setting page A first is a matter of fiddling with all the pointers in the right order. Of course, if it's already the FIRST_Z_PAGE, we're done.

42 ⟨*Set page first* 42⟩≡ (207)

```
set_page_first:
    SUBROUTINE

    CMP     FIRST_Z_PAGE
    BEQ     .end
    LDX     FIRST_Z_PAGE            ; prev_first = FIRST_Z_PAGE
    STA     FIRST_Z_PAGE            ; FIRST_Z_PAGE = A

    TAY                             ; SCRATCH2L = NEXT_PAGE_TABLE[FIRST_Z_PAGE]
    LDA     (NEXT_PAGE_TABLE),Y
    STA     SCRATCH2
    TXA                             ; NEXT_PAGE_TABLE[FIRST_Z_PAGE] = prev_first
    STA     (NEXT_PAGE_TABLE),Y

    LDA     (PREV_PAGE_TABLE),Y     ; SCRATCH2H = PREV_PAGE_TABLE[FIRST_Z_PAGE]
    STA     SCRATCH2+1
    LDA     #$FF                    ; PREV_PAGE_TABLE[FIRST_Z_PAGE] = #$FF
    STA     (PREV_PAGE_TABLE),Y
    LDY     SCRATCH2+1
    LDA     SCRATCH2
    STA     (NEXT_PAGE_TABLE),Y     ; NEXT_PAGE_TABLE[SCRATCH2H] = SCRATCH2L
    TXA
    TAY
    LDA     FIRST_Z_PAGE
    STA     (PREV_PAGE_TABLE),Y     ; PREV_PAGE_TABLE[prev_first] = FIRST_Z_PAGE
    LDA     SCRATCH2
    CMP     #$FF
    BEQ     .set_last_z_page
    TAY
    LDA     SCRATCH2+1
    STA     (PREV_PAGE_TABLE),Y     ; PREV_PAGE_TABLE[SCRATCH2L] = SCRATCH2H

.end:
    RTS

.set_last_z_page:
    LDA     SCRATCH2+1              ; LAST_Z_PAGE = SCRATCH2H
    STA     LAST_Z_PAGE
    RTS
```

Defines:
set_page_first, used in chunks 38, 40, and 43.

Uses FIRST_Z_PAGE 204, LAST_Z_PAGE 204, NEXT_PAGE_TABLE 204, PREV_PAGE_TABLE 204, and SCRATCH2 204.

The `get_next_code_byte2` routine is identical to `get_next_code_byte`, except that it uses a second set of `Z_PC` variables: `Z_PC2`, `ZCODE_PAGE_VALID2`, `ZCODE_PAGE_ADDR2`, and `PAGE_TABLE_INDEX2`.

Note that the three bytes of `Z_PC2` are not stored in memory in the same order as `Z_PC`, which is why we separate out the bytes into `Z_PC2_HH`, `Z_PC2_H`, and `Z_PC2_L`.

43 ⟨*Get next code byte 2* 43⟩≡ (207)

```
get_next_code_byte2:
    SUBROUTINE

    LDA     ZCODE_PAGE_VALID2
    BEQ     .zcode_page_invalid
    LDY     Z_PC2_L
    LDA     (ZCODE_PAGE_ADDR2),Y
    INY
    STY     Z_PC2_L
    BEQ     .invalidate_zcode_page
    RTS

.invalidate_zcode_page:
    LDY     #$00
    STY     ZCODE_PAGE_VALID2
    INC     Z_PC2_H
    BNE     .end
    INC     Z_PC2_HH

.end:
    RTS

.zcode_page_invalid:
    LDA     Z_PC2_HH
    BNE     .find_pc_page_in_page_table
    LDA     Z_PC2_H
    CMP     NUM_IMAGE_PAGES
    BCC     .set_page_addr

.find_pc_page_in_page_table:
    LDA     Z_PC2_H
    STA     SCRATCH2
    LDA     Z_PC2_HH
    STA     SCRATCH2+1
    JSR     find_index_of_page_table
    STA     PAGE_TABLE_INDEX2
    BCS     .not_found_in_page_table

.set_page_first:
    JSR     set_page_first
    CLC
```

```
    LDA     PAGE_TABLE_INDEX2
    ADC     NUM_IMAGE_PAGES

.set_page_addr:
    CLC
    ADC     Z_HEADER_ADDR+1
    STA     ZCODE_PAGE_ADDR2+1
    LDA     #$00
    STA     ZCODE_PAGE_ADDR2
    LDA     #$FF
    STA     ZCODE_PAGE_VALID2
    JMP     get_next_code_byte2

.not_found_in_page_table:
    CMP     PAGE_TABLE_INDEX
    BNE     .read_from_disk
    LDA     #$00
    STA     ZCODE_PAGE_VALID

.read_from_disk:
    LDA     AFTER_Z_IMAGE_ADDR
    STA     SCRATCH2
    LDA     AFTER_Z_IMAGE_ADDR+1
    STA     SCRATCH2+1
    LDA     PAGE_TABLE_INDEX2
    CLC
    ADC     SCRATCH2+1
    STA     SCRATCH2+1
    LDA     Z_PC2_H
    STA     SCRATCH1
    LDA     Z_PC2_HH
    STA     SCRATCH1+1
    JSR     read_from_sector
    BCC     .good_read
    JMP     main

.good_read:
    LDY     PAGE_TABLE_INDEX2
    LDA     Z_PC2_H
    STA     (PAGE_L_TABLE),Y
    LDA     Z_PC2_HH
    STA     (PAGE_H_TABLE),Y
    TYA
    JMP     .set_page_first
```

Defines:
get_next_code_byte2, used in chunks 45a and 166a.

Uses AFTER_Z_IMAGE_ADDR 204, NUM_IMAGE_PAGES 204, PAGE_H_TABLE 204, PAGE_L_TABLE 204, PAGE_TABLE_INDEX 204, PAGE_TABLE_INDEX2 204, SCRATCH1 204, SCRATCH2 204, ZCODE_PAGE_ADDR2 204, ZCODE_PAGE_VALID 204, ZCODE_PAGE_VALID2 204, Z_PC2_H 204, Z_PC2_HH 204, Z_PC2_L 204, find_index_of_page_table 41, main 26a, read_from_sector 107, and set_page_first 42.

That routine is used in get_next_code_word, which simply gets a 16-bit bigendian value at Z_PC2 and stores it in SCRATCH2.

45a ⟨*Get next code word* 45a⟩≡ (207)

```
get_next_code_word:
    SUBROUTINE

    JSR         get_next_code_byte2
    PHA
    JSR         get_next_code_byte2
    STA         SCRATCH2
    PLA
    STA         SCRATCH2+1
    RTS
```

Defines:
get_next_code_word, used in chunks 61 and 165b.
Uses SCRATCH2 204 and get_next_code_byte2 43.

The load_address routine copies SCRATCH2 to Z_PC2.

45b ⟨*Load address* 45b⟩≡ (207)

```
load_address:
    SUBROUTINE

    LDA         SCRATCH2
    STA         Z_PC2_L
    LDA         SCRATCH2+1
    STA         Z_PC2_H
    LDA         #$00
    STA         Z_PC2_HH
```

Defines:
load_address, used in chunks 137, 165b, 166a, and 185b.
Uses SCRATCH2 204, Z_PC2_H 204, Z_PC2_HH 204, and Z_PC2_L 204.

The `load_packed_address` routine multiplies `SCRATCH2` by 2 and stores the result in `Z_PC2`.

46 ⟨*Load packed address* 46⟩≡ (207)

```
invalidate_zcode_page2:
    SUBROUTINE

    LDA     #$00
    STA     ZCODE_PAGE_VALID2
    RTS

load_packed_address:
    SUBROUTINE

    LDA     SCRATCH2
    ASL
    STA     Z_PC2_L
    LDA     SCRATCH2+1
    ROL
    STA     Z_PC2_H
    LDA     #$00
    ROL
    STA     Z_PC2_HH
    JMP     invalidate_zcode_page2
```

Defines:
`invalidate_zcode_page2`, never used.
`load_packed_address`, used in chunks 65 and 186c.

Uses `SCRATCH2` 204, `ZCODE_PAGE_VALID2` 204, `Z_PC2_H` 204, `Z_PC2_HH` 204, and `Z_PC2_L` 204.

Chapter 7

I/O

7.1 Strings and output

7.1.1 The Apple II text screen

The `cout_string` routine stores a pointer to the ASCII string to print in `SCRATCH2`, and the number of characters to print in the `X` register. It uses the `COUT1` routine to output characters to the screen.

Apple II Monitors Peeled describes `COUT1` as writing the byte in the `A` register to the screen at cursor position `CV, CH`, using `INVFLG` and supporting cursor movement.

The difference between `COUT` and `COUT1` is that `COUT1` always prints to the screen, while `COUT` prints to whatever device is currently set as the output (e.g. a modem).

See also Apple II Reference Manual (Apple, 1979) page 61 for an explanation of these routines.

The logical-or with `#$80` sets the high bit, which causes `COUT1` to output normal characters. Without it, the characters would be in inverse text.

47 ⟨*Output string to console* 47⟩≡ (207)

```
cout_string:
    SUBROUTINE

    LDY      #$00

.loop:
```

```
      LDA     (SCRATCH2),Y
      ORA     #$80
      JSR     COUT1
      INY
      DEX
      BNE     .loop
      RTS
```

Defines:
  cout_string, used in chunks 54, 69, and 145.
Uses COUT1 203 and SCRATCH2 204.

The home routine calls the ROM HOME routine, which clears the scroll window and sets the cursor to the top left corner of the window. This routine, however, also loads CURR_LINE with the top line of the window.

48 ⟨*Home* 48⟩≡ (207)

```
  home:
      SUBROUTINE

      JSR     HOME
      LDA     WNDTOP
      STA     CURR_LINE
      RTS
```

Defines:
  home, used in chunks 49 and 143.
Uses CURR_LINE 204, HOME 203, and WNDTOP 203.

The `reset_window` routine sets the top left and bottom right of the screen scroll window to their full-screen values, sets the input prompt character to `>`, resets the inverse flag to `#$FF` (do not invert), then calls `home` to reset the cursor.

49 ⟨*Reset window* 49⟩≡ (207)

```
reset_window:
    SUBROUTINE

    LDA     #1
    STA     WNDTOP
    LDA     #0
    STA     WNDLFT
    LDA     #40
    STA     WNDWDTH
    LDA     #24
    STA     WNDBTM
    LDA     #$3E        ; '>'
    STA     PROMPT
    LDA     #$FF
    STA     INVFLG
    JSR     home
    RTS
```

Defines:
`reset_window`, used in chunk 29a.

Uses `INVFLG` 203, `PROMPT` 203, `WNDBTM` 203, `WNDLFT` 203, `WNDTOP` 203, `WNDWDTH` 203, and `home` 48.

7.1.2 The text buffer

When printing to the screen, Zork breaks lines between words. To do this, we buffer characters into the `BUFF_AREA`, which starts at address `$0200`. The offset into the area to put the next character into is in `BUFF_END`.

The `dump_buffer_to_screen` routine dumps the current buffer line to the screen, and then zeros `BUFF_END`.

50 ⟨*Dump buffer to screen* 50⟩≡ (207)

```
dump_buffer_to_screen:
    SUBROUTINE

    LDX     #$00

.loop:
    CPX     BUFF_END
    BEQ     .done
    LDA     BUFF_AREA,X
    JSR     COUT1
    INX
    JMP     .loop

.done:
    LDX     #$00
    STX     BUFF_END
    RTS
```

Defines:
`dump_buffer_to_screen`, used in chunks 53 and 69.
Uses `BUFF_AREA` 204, `BUFF_END` 204, and `COUT1` 203.

Zork also has the option to send all output to the printer, and the `dump_buffer_to_printer` routine is the printer version of `dump_buffer_to_screen`.

Output to the printer involves temporarily changing `CSW` (initially `COUT1`) to the printer output routine at `PRINTER_CSW`, calling `COUT` with the characters to print, then restoring `CSW`. Note that we call `COUT`, not `COUT1`.

See Apple II Reference Manual (Apple, 1979) page 61 for an explanation of these routines.

If the printer hasn't yet been initialized, we send the command string `ctrl-I80N`, which according to the Apple II Parallel Printer Interface Card Installation and Operation Manual, sets the printer to output 80 characters per line.

There is one part of initialization which isn't clear. It stores `#$91`, corresponding to character `Q`, into a screen memory hole at `$0779`. The purpose of doing this is not known.

See Understanding the Apple //e (Sather, 1985) figure 5.5 for details on screen holes.

See Apple II Reference Manual (Apple, 1979) page 82 for a possible explanation, where `$0779` is part of SCRATCHpad RAM for slot 1, which is typically where the printer card would be placed. Maybe writing `#$91` to `$0779` was necessary to enable command mode for certain cards.

51 ⟨*Dump buffer to printer* 51⟩≡ (207)

```
printer_card_initialized_flag:
    BYTE     00

dump_buffer_to_printer:
    SUBROUTINE

    LDA      CSW
    PHA
    LDA      CSW+1
    PHA
    LDA      PRINTER_CSW
    STA      CSW
    LDA      PRINTER_CSW+1
    STA      CSW+1
    LDX      #$00
    LDA      printer_card_initialized_flag
    BNE      .loop
    INC      printer_card_initialized_flag

.printer_set_80_column_output:
    LDA      #$09      ; ctrl-I
    JSR      COUT
    LDA      #$91      ; 'Q'
```

```
        STA     $0779       ; Scratchpad RAM for slot 1.
        LDA     #$B8        ; '8'
        JSR     COUT
        LDA     #$B0        ; '0'
        JSR     COUT
        LDA     #$CE        ; 'N'
        JSR     COUT

    .loop:
        CPX     BUFF_END
        BEQ     .done
        LDA     BUFF_AREA,X
        JSR     COUT
        INX
        JMP     .loop

    .done:
        LDA     CSW
        STA     PRINTER_CSW
        LDA     CSW+1
        STA     PRINTER_CSW+1
        PLA
        STA     CSW+1
        PLA
        STA     CSW
        RTS
```

Defines:
  `dump_buffer_to_printer`, used in chunks 53 and 71.
  `printer_card_initialized_flag`, never used.
Uses `BUFF_AREA` 204, `BUFF_END` 204, `COUT` 203, `CSW` 203, and `PRINTER_CSW` 204.

Tying these two routines together is `dump_buffer_line`, which dumps the current buffer line to the screen, and optionally the printer, depending on the printer output flag stored in bit `0` of offset `#$11` in the Z-machine header. Presumably this bit is set (in the Z-code itself) when you type `SCRIPT` on the Zork command line, and unset when you type `UNSCRIPT`.

53 ⟨*Dump buffer line* 53⟩≡ (207)

```
dump_buffer_line:
    SUBROUTINE

    LDY      #$11
    LDA      (Z_HEADER_ADDR),Y
    AND      #$01
    BEQ      .skip_printer
    JSR      dump_buffer_to_printer

.skip_printer:
    JSR      dump_buffer_to_screen
    RTS
```

Defines:
`dump_buffer_line`, used in chunks 55a, 69, 71, 145, 147a, and 148.
Uses `dump_buffer_to_printer` 51 and `dump_buffer_to_screen` 50.

The dump_buffer_with_more routine dumps the buffered line, but first, we check if we've reached the bottom of the screen by comparing CURR_LINE >= WNDBTM. If true, we print [MORE] in inverse text, wait for the user to hit a character, set CURR_LINE to WNDTOP + 1, and continue.

54 ⟨*Dump buffer with more* 54⟩≡ (207) 55a ▷

```
string_more:
    DC      "[MORE]"

dump_buffer_with_more:
    SUBROUTINE

    INC     CURR_LINE
    LDA     CURR_LINE
    CMP     WNDBTM
    BCC     .good_to_go    ; haven't reached bottom of screen yet

    STOW    string_more, SCRATCH2
    LDX     #6

    LDA     #$3F
    STA     INVFLG
    JSR     cout_string    ; print [MORE] in inverse text

    LDA     #$FF
    STA     INVFLG

    JSR     RDKEY      ; wait for keypress
    LDA     CH
    SEC
    SBC     #$06
    STA     CH             ; move cursor back 6
    JSR     CLREOL     ; and clear the line
    LDA     WNDTOP
    STA     CURR_LINE
    INC     CURR_LINE      ; start at top of screen

.good_to_go:
```

Defines:
dump_buffer_with_more, used in chunks 57, 58b, 143, 145, 147, 148, 200b, and 201.

Uses CH 203, CLREOL 203, CURR_LINE 204, INVFLG 203, RDKEY 203, SCRATCH2 204, STOW 10, WNDBTM 203, WNDTOP 203, and cout_string 47.

Next, we call `dump_buffer_line` to output the buffer to the screen. If we haven't yet reached the end of the line, then output a newline character to the screen.

55a ⟨*Dump buffer with more* 54⟩+≡ (207) ◁54 55b▷

```
        LDA     BUFF_END
        PHA
        JSR     dump_buffer_line
        PLA
        CMP     WNDWDTH
        BEQ     .skip_newline
        LDA     #$8D
        JSR     COUT1

    .skip_newline:
```

Uses `BUFF_END` 204, `COUT1` 203, `WNDWDTH` 203, and `dump_buffer_line` 53.

Next, we check if we are also outputting to the printer. If so, we output a newline to the printer as well. Note that we've already output the line to the printer in `dump_buffer_line`, so we only need to output a newline here.

55b ⟨*Dump buffer with more* 54⟩+≡ (207) ◁55a 56▷

```
        LDY     #$11
        LDA     (Z_HEADER_ADDR),Y
        AND     #$01
        BEQ     .reset_buffer_end

        LDA     CSW
        PHA
        LDA     CSW+1
        PHA
        LDA     PRINTER_CSW
        STA     CSW
        LDA     PRINTER_CSW+1
        STA     CSW+1

        LDA     #$8D
        JSR     COUT

        LDA     CSW
        STA     PRINTER_CSW
        LDA     CSW+1
        STA     PRINTER_CSW+1
        PLA
        STA     CSW+1
        PLA
        STA     CSW

    .reset_buffer_end:
```

Uses `COUT` 203, `CSW` 203, and `PRINTER_CSW` 204.

The last step is to set BUFF_END to zero.

56 ⟨*Dump buffer with more* 54⟩+≡ (207) ◁55b

```
        LDX     #$00
        JMP     buffer_char_set_buffer_end
```

Uses buffer_char_set_buffer_end 57.

The high-level routine `buffer_char` places the ASCII character in the `A` register into the end of the buffer.

If the character was a newline, then we tail-call to `dump_buffer_with_more` to dump the buffer to the output and return. Calling `dump_buffer_with_more` also resets `BUFF_END` to zero.

Otherwise, the character is first converted to uppercase if it is lowercase, then stored in the buffer and, if we haven't yet hit the end of the row, we increment `BUFF_END` and then return.

Control characters (those under `#$20`) are not put in the buffer, and simply ignored.

57 ⟨*Buffer a character* 57⟩≡ (207) 58a ▷

```
buffer_char:
    SUBROUTINE

    LDX      BUFF_END
    CMP      #$0D
    BNE      .not_0D
    JMP      dump_buffer_with_more

.not_0D:
    CMP      #$20
    BCC      buffer_char_set_buffer_end
    CMP      #$60
    BCC      .store_char
    CMP      #$80
    BCS      .store_char
    SEC
    SBC      #$20              ; converts to uppercase

.store_char:
    ORA      #$80              ; sets as normal text
    STA      BUFF_AREA,X
    CPX      WNDWDTH
    BCS      .hit_right_limit
    INX

buffer_char_set_buffer_end:
    STX      BUFF_END
    RTS

.hit_right_limit:
```

Defines:
`buffer_char`, used in chunks 59b, 66a, 67c, 69, 103, 104, 144a, 146, 182b, 184b, and 185c.
`buffer_char_set_buffer_end`, used in chunk 56.

Uses `BUFF_AREA` 204, `BUFF_END` 204, `WNDWDTH` 203, and `dump_buffer_with_more` 54.

If we have hit the end of a row, we're going to put the word we just wrote onto the next line.

To do that, we search for the position of the last space in the buffer, or if there wasn't any space, we just use the position of the end of the row.

58a ⟨*Buffer a character* 57⟩+≡ (207) ◁57 58b▷

```
    LDA     #$A0  ; normal space

.loop:
    CMP     BUFF_AREA,X
    BEQ     .endloop
    DEX
    BNE     .loop
    LDX     WNDWDTH

.endloop:
```

Uses BUFF_AREA 204 and WNDWDTH 203.

Now that we've found the position to break the line at, we dump the buffer up until that position using dump_buffer_with_more, which also resets BUFF_END to zero.

58b ⟨*Buffer a character* 57⟩+≡ (207) ◁58a 59a▷

```
    STX     BUFF_LINE_LEN
    STX     BUFF_END
    JSR     dump_buffer_with_more
```

Uses BUFF_END 204, BUFF_LINE_LEN 204, and dump_buffer_with_more 54.

Next, we increment BUFF_LINE_LEN to skip past the space. If we're past the window width though, we take the last character we added, move it to the end of the buffer (which should be the beginning of the buffer), increment BUFF_END, then we increment BUFF_LINE_LEN.

59a ⟨*Buffer a character* 57⟩+≡ (207) ◁58b

```
.increment_length:
    INC     BUFF_LINE_LEN
    LDX     BUFF_LINE_LEN
    CPX     WNDWDTH
    BCC     .move_last_char
    BEQ     .move_last_char
    RTS

.move_last_char:
    LDA     BUFF_AREA,X
    LDX     BUFF_END
    STA     BUFF_AREA,X
    INC     BUFF_END
    LDX     BUFF_LINE_LEN
    JMP     .increment_length
```

Uses BUFF_AREA 204, BUFF_END 204, BUFF_LINE_LEN 204, and WNDWDTH 203.

We can print an ASCII string with the print_ascii_string routine. It takes the length of the string in the X register, and the address of the string in SCRATCH2. It calls buffer_char to buffer each character in the string.

59b ⟨*Print ASCII string* 59b⟩≡ (207)

```
print_ascii_string:
    SUBROUTINE

    STX     SCRATCH3
    LDY     #$00
    STY     SCRATCH3+1

.loop:
    LDY     SCRATCH3+1
    LDA     (SCRATCH2),Y
    JSR     buffer_char
    INC     SCRATCH3+1
    DEC     SCRATCH3
    BNE     .loop
    RTS
```

Defines:
print_ascii_string, used in chunks 143, 145, 147a, 148, and 201.

Uses SCRATCH2 204, SCRATCH3 204, and buffer_char 57.

7.1.3 Z-coded strings

For how strings and characters are encoded, see section 3 of the Z-machine standard.

The alphabet shifts are stored in SHIFT_ALPHABET for a one-character shift, and SHIFT_LOCK_ALPHABET for a locked shift. The routine get_alphabet gets the alphabet to use, accounting for shifts.

60 ⟨*Get alphabet* 60⟩≡ (207)

```
get_alphabet:
    LDA     SHIFT_ALPHABET
    BPL     .remove_shift
    LDA     LOCKED_ALPHABET
    RTS

.remove_shift:
    LDY     #$FF
    STY     SHIFT_ALPHABET
    RTS
```

Defines:
get_alphabet, used in chunks 63a and 64.
Uses LOCKED_ALPHABET 204 and SHIFT_ALPHABET 204.

Since z-characters are encoded three at a time in two consecutive bytes in z-code, there's a state machine which determines where we are in the decompression. The state is stored in `ZDECOMPRESS_STATE`.

If `ZDECOMPRESS_STATE` is `0`, then we need to load the next two bytes from z-code and extract the first character. If `ZDECOMPRESS_STATE` is `1`, then we need to extract the second character. If `ZDECOMPRESS_STATE` is `2`, then we need to extract the third character. And finally if `ZDECOMPRESS_STATE` is `-1`, then we've reached the end of the string.

The z-character is returned in the `A` register. Furthermore, the carry is set when requesting the next character, but we've already reached the end of the string. Otherwise the carry is cleared.

61 ⟨*Get next zchar* 61⟩≡ (207)

```
get_next_zchar:
    LDA      ZDECOMPRESS_STATE
    BPL      .check_for_char_1
    SEC
    RTS

.check_for_char_1:
    BNE      .check_for_char_2
    INC      ZDECOMPRESS_STATE
    JSR      get_next_code_word
    LDA      SCRATCH2
    STA      ZCHARS_L
    LDA      SCRATCH2+1
    STA      ZCHARS_H
    LDA      ZCHARS_H
    LSR
    LSR
    AND      #$1F
    CLC
    RTS

.check_for_char_2:
    SEC
    SBC      #$01
    BNE      .check_for_last
    LDA      #$02
    STA      ZDECOMPRESS_STATE
    LDA      ZCHARS_H
    LSR
    LDA      ZCHARS_L
    ROR
    TAY
    LDA      ZCHARS_H
    LSR
    LSR
```

```
        TYA
        ROR
        LSR
        LSR
        LSR
        AND     #$1F
        CLC
        RTS

    .check_for_last:
        LDA     #$00
        STA     ZDECOMPRESS_STATE
        LDA     ZCHARS_H
        BPL     .get_char_3
        LDA     #$FF
        STA     ZDECOMPRESS_STATE

    .get_char_3:
        LDA     ZCHARS_L
        AND     #$1F
        CLC
        RTS
```

Defines:
get_next_zchar, used in chunks 63a, 65, and 68a.
Uses SCRATCH2 204, ZCHARS_H 204, ZCHARS_L 204, ZDECOMPRESS_STATE 204, and get_next_code_word 45a.

The print_zstring routine prints the z-encoded string at Z_PC2 to the screen. It uses get_next_zchar to get the next z-character, and handles alphabet shifts.

We first initialize the shift state.

62 ⟨*Print zstring* 62⟩≡ (207) 63a ▷

```
    print_zstring:
        SUBROUTINE

        LDA     #$00
        STA     LOCKED_ALPHABET
        STA     ZDECOMPRESS_STATE
        STOB    #$FF, SHIFT_ALPHABET
```

Defines:
print_zstring, used in chunks 65, 68b, 137, and 159b.
Uses LOCKED_ALPHABET 204, SHIFT_ALPHABET 204, STOB 11b, and ZDECOMPRESS_STATE 204.

Next, we loop through the z-string, getting each z-character. We have to handle special z-characters separately.

z-character 0 is always a space.

z-character 1 means to look at the next z-character and use it as an index into the abbreviation table, printing that string.

z-characters 2 and 3 shifts the alphabet forwards (A0 to A1 to A2 to A0) and backwards (A0 to A2 to A1 to A0) respectively.

z-characters 4 and 5 shift-locks the alphabet.

All other characters will get translated to the ASCII character using the current alphabet.

63a ⟨*Print zstring* 62⟩+≡ (207) ◁62

```
  .loop:
      JSR       get_next_zchar
      BCC       .not_end
      RTS

  .not_end:
      STA       SCRATCH3
      BEQ       .space                  ; z-char 0?
      CMP       #$01
      BEQ       .abbreviation           ; z-char 1?
      CMP       #$04
      BCC       .shift_alphabet         ; z-char 2 or 3?
      CMP       #$06
      BCC       .shift_lock_alphabet    ; z-char 4 or 5?
      JSR       get_alphabet

      ; fall through to print the z-character
  ⟨Print the zchar 66a⟩
```

Uses SCRATCH3 204, get_alphabet 60, and get_next_zchar 61.

63b ⟨*Printing a space* 63b⟩≡ (207)

```
  .space:
      LDA       #$20
      JMP       .printchar
```

Defines:
.space, never used.

64 ⟨*Shifting alphabets 64*⟩≡ (207)
```
  .shift_alphabet:
      JSR       get_alphabet
      CLC
      ADC       #$02
      ADC       SCRATCH3
      JSR       A_mod_3
      STA       SHIFT_ALPHABET
      JMP       .loop

  .shift_lock_alphabet:
      JSR       get_alphabet
      CLC
      ADC       SCRATCH3
      JSR       A_mod_3
      STA       LOCKED_ALPHABET
      JMP       .loop
```
Defines:
  `.shift_alphabet`, never used.
  `.shift_lock_alphabet`, never used.
Uses `A_mod_3` 102, `LOCKED_ALPHABET` 204, `SCRATCH3` 204, `SHIFT_ALPHABET` 204, and `get_alphabet` 60.

When printing an abbrevation, we multiply the z-character by 2 to get an address index into Z_ABBREV_TABLE. The address from the table is then stored in SCRATCH2, and we recurse into print_zstring to print the abbreviation. This involves saving and restoring the current decompress state.

65 ⟨*Printing an abbreviation* 65⟩≡ (207)

```
.abbreviation:
    JSR     get_next_zchar
    ASL
    ADC     #$01
    TAY
    LDA     (Z_ABBREV_TABLE),Y
    STA     SCRATCH2
    DEY
    LDA     (Z_ABBREV_TABLE),Y
    STA     SCRATCH2+1

    ; Save the decompress state

    LDA     LOCKED_ALPHABET
    PHA
    LDA     ZDECOMPRESS_STATE
    PHA
    LDA     ZCHARS_L
    PHA
    LDA     ZCHARS_H
    PHA
    LDA     Z_PC2_L
    PHA
    LDA     Z_PC2_H
    PHA
    LDA     Z_PC2_HH
    PHA

    JSR     load_packed_address
    JSR     print_zstring

    ; Restore the decompress state

    PLA
    STA     Z_PC2_HH
    PLA
    STA     Z_PC2_H
    PLA
    STA     Z_PC2_L
    LDA     #$00
    STA     ZCODE_PAGE_VALID2
    PLA
    STA     ZCHARS_H
    PLA
```

```
    STA     ZCHARS_L
    PLA
    STA     ZDECOMPRESS_STATE
    PLA
    STA     LOCKED_ALPHABET
    LDA     #$FF             ; Resets any temporary shift
    STA     SHIFT_ALPHABET
    JMP     .loop
```

Defines:
`.abbreviation`, never used.

Uses `LOCKED_ALPHABET` 204, `SCRATCH2` 204, `SHIFT_ALPHABET` 204, `ZCHARS_H` 204, `ZCHARS_L` 204, `ZCODE_PAGE_VALID2` 204, `ZDECOMPRESS_STATE` 204, `Z_ABBREV_TABLE` 204, `Z_PC2_H` 204, `Z_PC2_HH` 204, `Z_PC2_L` 204, `get_next_zchar` 61, `load_packed_address` 46, and `print_zstring` 62.

If we are on alphabet `0`, then we print the ASCII character directly by adding `#$5B`. Remember that we are handling 26 z-characters `6-31`, so the ASCII characters will be `a-z`.

66a ⟨*Print the zchar* 66a⟩≡ (63a) 66b▷

```
    ORA     #$00
    BNE     .check_for_alphabet_A1
    LDA     #$5B

.add_ascii_offset:
    CLC
    ADC     SCRATCH3

.printchar:
    JSR     buffer_char
    JMP     .loop
```

Uses `SCRATCH3` 204 and `buffer_char` 57.

Alphabet `1` handles uppercase characters `A-Z`, so we add `#$3B` to the z-char.

66b ⟨*Print the zchar* 66a⟩+≡ (63a) ◁66a 67b▷

```
.check_for_alphabet_A1:
    CMP     #$01
    BNE     .map_ascii_for_A2
    LDA     #$3B
    JMP     .add_ascii_offset
```

Defines:
`.check_for_alphabet_A1`, never used.

Alphabet 2 is more complicated because it doesn't map consecutively onto ASCII characters.

z-character 6 in alphabet 2 means that the two subsequent z-characters specify a ten-bit ZSCII character code: the next z-character gives the top 5 bits and the one after the bottom 5. However, in this version of the interpreter, only 8 bits are kept, and these are simply ASCII values.

z-character 7 causes a CRLF to be output.

Otherwise, we map the z-character to the ASCII character using the a2_table table.

67a ⟨*A2 table* 67a⟩≡ (207)

```
  a2_table:
      DC        "0123456789.,!?_#"
      DC        '"
      DC        "'/\-:()"
```

Defines:
a2_table, used in chunks 67b and 88b.

67b ⟨*Print the zchar* 66a⟩+≡ (63a) ◁66b

```
  .map_ascii_for_A2:
      LDA       SCRATCH3
      SEC
      SBC       #$07
      BCC       .z10bits
      BEQ       .crlf
      TAY
      DEY
      LDA       a2_table,Y
      JMP       .printchar
```

Defines:
.map_ascii_for_A2, never used.
Uses SCRATCH3 204 and a2_table 67a.

67c ⟨*Printing a CRLF* 67c⟩≡ (207)

```
  .crlf:
      LDA       #$0D
      JSR       buffer_char
      LDA       #$0A
      JMP       .printchar
```

Defines:
.crlf, never used.
Uses buffer_char 57.

68a ⟨*Printing a 10-bit ZSCII character* 68a⟩≡ (207)
```
.z10bits:
    JSR     get_next_zchar
    ASL
    ASL
    ASL
    ASL
    ASL
    PHA
    JSR     get_next_zchar
    STA     SCRATCH3
    PLA
    ORA     SCRATCH3
    JMP     .printchar
```
Defines:
  .z10bits, never used.
Uses SCRATCH3 204 and get_next_zchar 61.

print_string_literal is a high-level routine that prints a string literal to the screen, where the string literal is in z-code at the current Z_PC.

68b ⟨*Printing a string literal* 68b⟩≡ (207)
```
print_string_literal:
    SUBROUTINE

    LDA     Z_PC
    STA     Z_PC2_L
    LDA     Z_PC+1
    STA     Z_PC2_H
    LDA     Z_PC+2
    STA     Z_PC2_HH
    LDA     #$00
    STA     ZCODE_PAGE_VALID2
    JSR     print_zstring
    LDA     Z_PC2_L
    STA     Z_PC
    LDA     Z_PC2_H
    STA     Z_PC+1
    LDA     Z_PC2_HH
    STA     Z_PC+2
    LDA     ZCODE_PAGE_VALID2
    STA     ZCODE_PAGE_VALID
    LDA     ZCODE_PAGE_ADDR2
    STA     ZCODE_PAGE_ADDR
    LDA     ZCODE_PAGE_ADDR2+1
    STA     ZCODE_PAGE_ADDR+1
    RTS
```
Uses ZCODE_PAGE_ADDR 204, ZCODE_PAGE_ADDR2 204, ZCODE_PAGE_VALID 204, ZCODE_PAGE_VALID2 204, Z_PC 204, Z_PC2_H 204, Z_PC2_HH 204, Z_PC2_L 204, and print_zstring 62.

The status line

Printing the status line involves saving the current cursor location, moving the cursor to the top left of the screen, setting inverse text, printing the current room name at column 0, printing the score at column 25, resetting inverse text, and then restoring the cursor location.

69 ⟨*Print status line* 69⟩≡ (207)

```
sScore:
    DC       "SCORE:"

print_status_line:
    SUBROUTINE

    JSR      dump_buffer_line
    LDA      CH
    PHA
    LDA      CV
    PHA
    LDA      #$00
    STA      CH
    STA      CV
    JSR      VTAB
    LDA      #$3F
    STA      INVFLG
    JSR      CLREOL

    LDA      #VAR_CURR_ROOM
    JSR      var_get
    JSR      print_obj_in_A
    JSR      dump_buffer_to_screen

    LDA      #25
    STA      CH
    LDA      #<sScore
    STA      SCRATCH2
    LDA      #>sScore
    STA      SCRATCH2+1
    LDX      #$06
    JSR      cout_string

    INC      CH
    LDA      #VAR_SCORE
    JSR      var_get
    JSR      print_number

    LDA      #'/
    JSR      buffer_char

    LDA      #VAR_MAX_SCORE
```

```
        JSR     var_get
        JSR     print_number
        JSR     dump_buffer_to_screen

        LDA     #$FF
        STA     INVFLG
        PLA
        STA     CV
        PLA
        STA     CH
        JSR     VTAB
        RTS
```

Defines:
print_status_line, used in chunk 73.
sScore, never used.

Uses CH 203, CLREOL 203, CV 203, INVFLG 203, SCRATCH2 204, VAR_CURR_ROOM 206b, VAR_MAX_SCORE 206b, VAR_SCORE 206b, VTAB 203, buffer_char 57, cout_string 47, dump_buffer_line 53, dump_buffer_to_screen 50, print_number 103, print_obj_in_A 137, and var_get 122.

7.1.4 Input

The `read_line` routine dumps whatever is in the output buffer to the output, then reads a line of input from the keyboard, storing it in the `BUFF_AREA` buffer. The buffer is terminated with a newline character.

The routine then checks if the transcript flag is set in the header, and if so, it dumps the buffer to the printer. The buffer is then truncated to the maximum number of characters allowed.

The routine then converts the characters to lowercase, and returns.

The `A` register will contain the number of characters in the buffer.

71 ⟨*Read line* 71⟩≡ (207)

```
read_line:
    SUBROUTINE

    JSR     dump_buffer_line
    LDA     WNDTOP
    STA     CURR_LINE
    JSR     GETLN1
    INC     CURR_LINE
    LDA     #$8D                ; newline
    STA     BUFF_AREA,X
    INX                         ; X = num of chars in input
    TXA
    PHA                         ; save X
    LDY     #HEADER_FLAGS2_OFFSET+1
    LDA     (Z_HEADER_ADDR),Y
    AND     #$01                ; Mask for transcript on
    BEQ     .continue
    TXA
    STA     BUFF_END
    JSR     dump_buffer_to_printer
    LDA     #$00
    STA     BUFF_END

.continue
    PLA                         ; restore num of chars in input
    LDY     #$00                ; truncate to max num of chars
    CMP     (OPERAND0),Y
    BCC     .continue2
    LDA     (OPERAND0),Y

.continue2:
    PHA                         ; save num of chars
    BEQ     .end
    TAX
```

```
.loop:
    LDA     BUFF_AREA,Y   ; convert A-Z to lowercase
    AND     #$7F
    CMP     #$41
    BCC     .continue3
    CMP     #$5B
    BCS     .continue3
    ORA     #$20

.continue3:
    INY
    STA     (OPERAND0),Y
    CMP     #$0D
    BEQ     .end
    DEX
    BNE     .loop

.end:
    PLA                        ; restore num of chars
    RTS
```

Defines:
`read_line`, used in chunk 73.

Uses `BUFF_AREA` 204, `BUFF_END` 204, `CURR_LINE` 204, `GETLN1` 203, `HEADER_FLAGS2_OFFSET` 206a, `OPERAND0` 204, `WNDTOP` 203, `dump_buffer_line` 53, and `dump_buffer_to_printer` 51.

7.1.5 Lexical parsing

After reading a line, the Z-machine needs to parse it into words and then look up those words in the dictionary. The `sread` instruction combines `read_line` with parsing.

`sread` redisplays the status line, then reads characters from the keyboard until a newline is entered. The characters are stored in the buffer at the z-address in `OPERAND0`, and parsed into the buffer at the z-address in `OPERAND1`.

Prior to this instruction, the first byte in the text buffer must contain the maximum number of characters to accept as input, minus 1.

After the line is read, the line is split into words (separated by the separators space, period, comma, question mark, carriage return, newline, tab, or formfeed), and each word is looked up in the dictionary.

The number of words parsed is written in byte 1 of the parse buffer, and then follows the tokens.

Each token is 4 bytes. The first two bytes are the address of the word in the dictionary (or 0 if not found), followed by the length of the word, followed by the index into the buffer where the word starts.

73 ⟨*Instruction sread* 73⟩≡ (207) 74a▷

```
instr_sread:
    SUBROUTINE

    JSR     print_status_line
    ADDW    OPERAND0, Z_HEADER_ADDR, OPERAND0   ; text buffer
    ADDW    OPERAND1, Z_HEADER_ADDR, OPERAND1   ; parse buffer
    JSR     read_line       ; SCRATCH3H = read_line() (input_count)
    STA     SCRATCH3+1
    LDA     #$00            ; SCRATCH3L = 0  (char count)
    STA     SCRATCH3
    LDY     #$01
    LDA     #$00            ; store 0 in the parse buffer + 1.
    STA     (OPERAND1),Y
    LDA     #$02
    STA     TOKEN_IDX
    LDA     #$01
    STA     INPUT_PTR
```

Defines:
`instr_sread`, used in chunk 109.

Uses `ADDW` 15c, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH3` 204, `print_status_line` 69, and `read_line` 71.

Loop:

We check the next two bytes in the parse buffer, and if they are the same, we are done.

74a ⟨*Instruction sread* 73⟩+≡ (207) ◁73 74b▷

```
  .loop_word:
      LDY      #$00          ; if parsebuf[0] == parsebuf[1] do_instruction
      LDA      (OPERAND1),Y
      INY
      CMP      (OPERAND1),Y
      BNE      .not_end1
      JMP      do_instruction
```

Uses OPERAND1 204 and do_instruction 112.

Also, if the char count and input buffer len are zero, we are done.

74b ⟨*Instruction sread* 73⟩+≡ (207) ◁74a 74c▷

```
  .not_end1:
      LDA      SCRATCH3+1    ; if input_count == char_count == 0 do_instruction
      ORA      SCRATCH3
      BNE      .not_end2
      JMP      do_instruction
```

Uses SCRATCH3 204 and do_instruction 112.

If the char count isn't yet 6, then we need more chars.

74c ⟨*Instruction sread* 73⟩+≡ (207) ◁74b 75a▷

```
  .not_end2:
      LDA      SCRATCH3      ; if char_count != 6 .not_min_compress_size
      CMP      #$06
      BNE      .not_min_compress_size
      JSR      skip_separators
```

Uses SCRATCH3 204 and skip_separators 79.

If the char count is 0, then we can initialize the 6-byte area in ZCHAR_SCRATCH1 with zero.

75a ⟨*Instruction sread* 73⟩+≡ (207) ◁74c 75b▷

```
  .not_min_compress_size:
      LDA       SCRATCH3
      BNE       .not_separator
      LDY       #$06
      LDX       #$00

  .clear:
      LDA       #$00
      STA       ZCHAR_SCRATCH1,X
      INX
      DEY
      BNE       .clear
```

Uses SCRATCH3 204 and ZCHAR_SCRATCH1 204.

Next we set up the token. Byte 3 in a token is the index into the text buffer where the word starts (INPUT_PTR). We then check if the character pointed to is a dictionary separator (which needs to be treated as a word) or a standard separator (which needs to be skipped over). And if the character is a standard separator, we increment the input pointer and decrement the input count and loop back.

75b ⟨*Instruction sread* 73⟩+≡ (207) ◁75a 76a▷

```
      LDA       INPUT_PTR           ; parsebuf[TOKEN_IDX+3] = INPUT_PTR
      LDY       TOKEN_IDX
      INY
      INY
      INY
      STA       (OPERAND1),Y
      LDY       INPUT_PTR           ; is_dict_separator(textbuf[INPUT_PTR])
      LDA       (OPERAND0),Y
      JSR       is_dict_separator
      BCS       .is_dict_separator
      LDY       INPUT_PTR           ; is_std_separator(textbuf[INPUT_PTR])
      LDA       (OPERAND0),Y
      JSR       is_std_separator
      BCC       .not_separator
      INC       INPUT_PTR           ; ++INPUT_PTR
      DEC       SCRATCH3+1          ; --input_count
      JMP       .loop_word
```

Uses OPERAND0 204, OPERAND1 204, SCRATCH3 204, is_dict_separator 80, and is_std_separator 80.

If char_count is zero, we have run out of characters, so we need to search through the dictionary with whatever we've collected in the ZCHAR_SCRATCH1 buffer.

We also check if the character is a separator, and if so, we again search through the dictionary with whatever we've collected in the ZCHAR_SCRATCH1 buffer.

Otherwise, we can store the character in the ZCHAR_SCRATCH1 buffer, increment the char count and input pointer and decrement the input count. Then loop back.

76a ⟨*Instruction sread* 73⟩+≡ (207) ◁75b 76b▷

```
  .not_separator:
      LDA     SCRATCH3+1
      BEQ     .search
      LDY     INPUT_PTR           ; is_separator(textbuf[INPUT_PTR])
      LDA     (OPERAND0),Y
      JSR     is_separator
      BCS     .search
      LDY     INPUT_PTR           ; ZCHAR_SCRATCH1[char_count] = textbuf[INPUT_PTR]
      LDA     (OPERAND0),Y
      LDX     SCRATCH3
      STA     ZCHAR_SCRATCH1,X
      DEC     SCRATCH3+1          ; --input_count
      INC     SCRATCH3            ; ++char_count
      INC     INPUT_PTR           ; ++INPUT_PTR
      JMP     .loop_word
```

Uses OPERAND0 204, SCRATCH3 204, ZCHAR_SCRATCH1 204, and is_separator 80.

If it's a dictionary separator, we store the character in the ZCHAR_SCRATCH1 buffer, increment the char count and input pointer and decrement the input count. Then we fall through to search.

76b ⟨*Instruction sread* 73⟩+≡ (207) ◁76a 77▷

```
  .is_dict_separator:
      STA     ZCHAR_SCRATCH1
      INC     SCRATCH3
      DEC     SCRATCH3+1
      INC     INPUT_PTR
```

Uses SCRATCH3 204, ZCHAR_SCRATCH1 204, and is_dict_separator 80.

To begin, if we haven't collected any characters, then just go back and loop again.

Next, we store the number of characters in the token into the current token at byte 2. Although we will only compare the first 6 characters, we store the number of input characters in the token.

77 ⟨*Instruction sread* 73⟩+≡ (207) ◁76b 78▷

```
.search:
    LDA      SCRATCH3
    BEQ      .loop_word
    LDA      SCRATCH3+1     ; Save input_count
    PHA
    LDY      TOKEN_IDX      ; parsebuf[TOKEN_IDX+2] = char_count
    INY
    INY
    LDA      SCRATCH3
    STA      (OPERAND1),Y
```

Uses OPERAND1 204 and SCRATCH3 204.

We then convert these characters into z-characters, which we then search through the dictionary for. We store the z-address of the found token (or zero if not found) into the token, and then loop back for the next word.

78 ⟨*Instruction sread* 73⟩+≡ (207) ◁77

```
    JSR     ascii_to_zchar
    JSR     match_dictionary_word
    LDY     TOKEN_IDX              ; parsebuf[TOKEN_IDX] = entry_addr
    LDA     SCRATCH1+1
    STA     (OPERAND1),Y
    INY
    LDA     SCRATCH1
    STA     (OPERAND1),Y

    INY                            ; TOKEN_IDX += 4
    INY
    INY
    STY     TOKEN_IDX

    LDY     #$01                   ; ++parsebuf[1]
    LDA     (OPERAND1),Y
    CLC
    ADC     #$01
    STA     (OPERAND1),Y

    PLA
    STA     SCRATCH3+1
    LDA     #$00
    STA     SCRATCH3
    JMP     .loop_word
```

Uses OPERAND1 204, SCRATCH1 204, SCRATCH3 204, ascii_to_zchar 81, and match_dictionary_word 91.

Separators

79 ⟨*Skip separators 79*⟩≡ (207)

```
skip_separators:
    SUBROUTINE

    LDA     SCRATCH3+1
    BNE     .not_end
    RTS

.not_end:
    LDY     INPUT_PTR
    LDA     (OPERAND0),Y
    JSR     is_separator
    BCC     .not_separator
    RTS

.not_separator:
    INC     INPUT_PTR
    DEC     SCRATCH3+1
    INC     SCRATCH3
    JMP     skip_separators
```

Defines:
  `skip_separators`, used in chunk 74c.
Uses `OPERAND0` 204, `SCRATCH3` 204, and `is_separator` 80.

80 ⟨*Separator checks* 80⟩≡ (207)

```
SEPARATORS_TABLE:
    DC      #$20, #$2E, #$2C, #$3F, #$0D, #$0A, #$09, #$0C

is_separator:
    SUBROUTINE

    JSR     is_dict_separator
    BCC     is_std_separator
    RTS

is_std_separator:
    SUBROUTINE

    LDY     #$00
    LDX     #$08

.loop:
    CMP     SEPARATORS_TABLE,Y
    BEQ     separator_found
    INY
    DEX
    BNE     .loop

separator_not_found:
    CLC
    RTS

separator_found:
    SEC
    RTS

is_dict_separator:
    SUBROUTINE

    PHA
    JSR     get_dictionary_addr
    LDY     #$00
    LDA     (SCRATCH2),Y
    TAX
    PLA

.loop:
    BEQ     separator_not_found
    INY
    CMP     (SCRATCH2),Y
    BEQ     separator_found
    DEX
    JMP     .loop
```

Defines:

SEPARATORS_TABLE, never used.
is_dict_separator, used in chunks 75b and 76b.
is_separator, used in chunks 76a and 79.
is_std_separator, used in chunk 75b.
separator_found, never used.
separator_not_found, never used.
Uses SCRATCH2 204 and get_dictionary_addr 90.

ASCII to Z-chars

The `ascii_to_zchar` routine converts the ASCII characters in the input buffer to z-characters.

We first set the `LOCKED_ALPHABET` shift to alphabet 0, and then clear the `ZCHAR_SCRATCH2` buffer with `05` (pad) zchars.

81 ⟨*ASCII to Zchar* 81⟩≡ (207) 82a ▷

```
ascii_to_zchar:
    SUBROUTINE

    LDA         #$00
    STA         LOCKED_ALPHABET
    LDX         #$00
    LDY         #$06

.clear:
    LDA         #$05
    STA         ZCHAR_SCRATCH2,X
    INX
    DEY
    BNE         .clear

    LDA         #$06
    STA         SCRATCH3+1          ; nchars = 6
    LDA         #$00
    STA         SCRATCH1            ; dest_index = 0
    STA         SCRATCH2            ; index = 0
```

Defines:
ascii_to_zchar, used in chunk 78.
Uses LOCKED_ALPHABET 204, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, and ZCHAR_SCRATCH2 204.

Next we loop over the input buffer, converting each character in ZCHAR_SCRATCH1 to a z-character. If the character is zero, we store a pad zchar.

82a ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁81 82b▷

```
  .loop:
      LDX      SCRATCH2           ; c = ZCHAR_SCRATCH1[index++]
      INC      SCRATCH2
      LDA      ZCHAR_SCRATCH1,X
      STA      SCRATCH3
      BNE      .continue
      LDA      #$05
      JMP      .store_zchar
```

Uses SCRATCH2 204, SCRATCH3 204, and ZCHAR_SCRATCH1 204.

We first check to see which alphabet the character is in. If the alphabet is the same as the alphabet we're currently locked into, then we go to .same_alphabet because we don't need to shift the alphabet.

82b ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁82a 83b▷

```
  .continue:
      LDA      SCRATCH1           ; save dest_index
      PHA
      LDA      SCRATCH3           ; alphabet = get_alphabet_for_char(c)
      JSR      get_alphabet_for_char
      STA      SCRATCH1
      CMP      LOCKED_ALPHABET
      BEQ      .same_alphabet
```

Uses LOCKED_ALPHABET 204, SCRATCH1 204, SCRATCH3 204, and get_alphabet_for_char 83a.

83a ⟨*Get alphabet for char* 83a⟩≡ (207)
```
get_alphabet_for_char:
    SUBROUTINE

    CMP     #$61
    BCC     .check_upper
    CMP     #$7B
    BCS     .check_upper
    LDA     #$00
    RTS

.check_upper:
    CMP     #$41
    BCC     .check_nonletter
    CMP     #$5B
    BCS     .check_nonletter
    LDA     #$01
    RTS

.check_nonletter:
    ORA     #$00
    BEQ     .return
    BMI     .return
    LDA     #$02

.return:
    RTS
```
Defines:
get_alphabet_for_char, used in chunks 82b, 83b, and 87a.

Otherwise we check the next character to see if it's in the same alphabet as the current character. If they're different, then we should shift the alphabet, not lock it.

83b ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁82b 84a▷
```
    LDX     SCRATCH2
    LDA     ZCHAR_SCRATCH1,X
    JSR     get_alphabet_for_char
    CMP     SCRATCH1
    BNE     .shift_alphabet
```
Uses SCRATCH1 204, SCRATCH2 204, ZCHAR_SCRATCH1 204, and get_alphabet_for_char 83a.

We then determine which direction to shift lock the alphabet to, store the shifting character into SCRATCH1+1, and set the locked alphabet to the new alphabet.

84a ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁83b 84b▷

```
    SEC                         ; shift_char = shift lock char (4 or 5)
    SBC     LOCKED_ALPHABET
    CLC
    ADC     #$03
    JSR     A_mod_3
    CLC
    ADC     #$03
    STA     SCRATCH1+1
    MOVB    SCRATCH1, LOCKED_ALPHABET   ; LOCKED_ALPHABET = alphabet
```

Uses A_mod_3 102, LOCKED_ALPHABET 204, MOVB 11b, and SCRATCH1 204.

Then we store the shift lock character into the destination buffer.

84b ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁84a 84c▷

```
    PLA                         ; restore dest_index
    STA     SCRATCH1
    LDA     SCRATCH1+1          ; ZCHAR_SCRATCH2[dest_index] = shift_char
    LDX     SCRATCH1
    STA     ZCHAR_SCRATCH2,X
    INC     SCRATCH1            ; ++dest_index
```

Uses SCRATCH1 204 and ZCHAR_SCRATCH2 204.

If we've run out of room in the destination buffer, then we simply go to compress the destination buffer and return. Otherwise we will add the character to the destination buffer by going to .same_alphabet.

84c ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁84b 86▷

```
    DEC     SCRATCH3+1          ; --nchars
    BNE     .add_shifted_char
    JMP     z_compress

.add_shifted_char:
    LDA     SCRATCH1            ; save dest_index
    PHA
    JMP     .same_alphabet
```

Uses SCRATCH1 204, SCRATCH3 204, and z_compress 85.

The z_compress routine takes the 6 z-characters in ZCHAR_SCRATCH2 and compresses them into 4 bytes.

85 ⟨*Z compress* 85⟩≡ (207)

```
z_compress:
    SUBROUTINE

    LDA     ZCHAR_SCRATCH2+1
    ASL
    ASL
    ASL
    ASL
    ROL     ZCHAR_SCRATCH2
    ASL
    ROL     ZCHAR_SCRATCH2
    LDX     ZCHAR_SCRATCH2
    STX     ZCHAR_SCRATCH2+1
    ORA     ZCHAR_SCRATCH2+2
    STA     ZCHAR_SCRATCH2
    LDA     ZCHAR_SCRATCH2+4
    ASL
    ASL
    ASL
    ASL
    ROL     ZCHAR_SCRATCH2+3
    ASL
    ROL     ZCHAR_SCRATCH2+3
    LDX     ZCHAR_SCRATCH2+3
    STX     ZCHAR_SCRATCH2+3
    ORA     ZCHAR_SCRATCH2+5
    STA     ZCHAR_SCRATCH2+2
    LDA     ZCHAR_SCRATCH2+3
    ORA     #$80
    STA     ZCHAR_SCRATCH2+3
    RTS
```

Defines:
z_compress, used in chunks 84c, 86, 87b, and 89.
Uses ZCHAR_SCRATCH2 204.

To temporarily shift the alphabet, we determine which character we need to use to shift it out of the current alphabet (LOCKED_ALPHABET), and put it in the destination buffer. Then, if we've run out of characters in the destination buffer, we simply go to compress the destination buffer and return.

86 ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁84c 87a▷

```
.shift_alphabet:
    LDA     SCRATCH1            ; shift_char = shift char (2 or 3)
    SEC
    SBC     LOCKED_ALPHABET
    CLC
    ADC     #$03
    JSR     A_mod_3
    TAX
    INX
    PLA
    STA     SCRATCH1            ; restore dest_index
    TXA                         ; ZCHAR_SCRATCH2[dest_index] = shift_char
    LDX     SCRATCH1
    STA     ZCHAR_SCRATCH2,X
    INC     SCRATCH1            ; ++dest_index
    DEC     SCRATCH3+1          ; --nchars
    BNE     .save_dest_index_and_same_alphabet

stretchy_z_compress:
    JMP     z_compress
```

Defines:
stretchy_z_compress, never used.

Uses A_mod_3 102, LOCKED_ALPHABET 204, SCRATCH1 204, SCRATCH3 204, ZCHAR_SCRATCH2 204, and z_compress 85.

If the character to save is lowercase, we can simply subtract `#$5B` such that 'a' = 6, and so on.

87a ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁86 87b▷

```
.save_dest_index_and_same_alphabet:
    LDA      SCRATCH1            ; save dest_index
    PHA

.same_alphabet:
    PLA
    STA      SCRATCH1            ; restore dest_index
    LDA      SCRATCH3
    JSR      get_alphabet_for_char
    SEC
    SBC      #$01                ; alphabet_minus_1 = case(c) - 1
    BPL      .not_lowercase
    LDA      SCRATCH3
    SEC
    SBC      #$5B                ; c -= 'a'-6
```

Uses `SCRATCH1` 204, `SCRATCH3` 204, and `get_alphabet_for_char` 83a.

Then we store the character in the destination buffer, and move on to the next character, unless the destination buffer is full, in which case we compress and return.

87b ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁87a 87c▷

```
.store_zchar:
    LDX      SCRATCH1            ; ZCHAR_SCRATCH2[dest_index] = c
    STA      ZCHAR_SCRATCH2,X
    INC      SCRATCH1            ; ++dest_index
    DEC      SCRATCH3+1          ; --nchars
    BEQ      .dest_full
    JMP      .loop

.dest_full:
    JMP      z_compress
```

Uses `SCRATCH1` 204, `SCRATCH3` 204, `ZCHAR_SCRATCH2` 204, and `z_compress` 85.

If the character was upper case, then we can subtract `#$3B` such that 'A' = 6, and so on, and then store the character in the same way.

87c ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁87b 88a▷

```
.not_lowercase:
    BNE      .not_alphabetic
    LDA      SCRATCH3
    SEC
    SBC      #$3B                ; c -= 'A'-6
    JMP      .store_zchar
```

Uses `SCRATCH3` 204.

Now if the character isn't upper or lower case, then it's a non-alphabetic character. We first search in the non-alphabetic table, and if found, we can store that character and continue.

88a ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁87c 89▷

```
.not_alphabetic:
    LDA      SCRATCH3
    JSR      search_nonalpha_table
    BNE      .store_zchar
```

Uses SCRATCH3 204 and search_nonalpha_table 88b.

88b ⟨*Search nonalpha table* 88b⟩≡ (207)

```
search_nonalpha_table:
    SUBROUTINE

    LDX      #$24

.loop:
    CMP      a2_table,X
    BEQ      .found
    DEX
    BPL      .loop
    LDY      #$00
    RTS

.found:
    TXA
    CLC
    ADC      #$08
    RTS
```

Defines:
search_nonalpha_table, used in chunk 88a.
Uses a2_table 67a.

If, however, the character is simply not representable in the z-characters, then we store a z-char newline (6), and, if there's still room in the destination buffer, we store the high 3 bits of the unrepresentable character and store it in the destination buffer, and, if there's still room, we take the low 5 bits and store that in the destination buffer.

This works because the newline character can never be a part of the input, so it serves here as an escaping character.

89 ⟨*ASCII to Zchar* 81⟩+≡ (207) ◁88a

```
    LDA     #$06              ; ZCHAR_SCRATCH2[dest_index] = 6
    LDX     SCRATCH1
    STA     ZCHAR_SCRATCH2,X
    INC     SCRATCH1          ; ++dest_index
    DEC     SCRATCH3+1        ; --nchars
    BEQ     z_compress

    LDA     SCRATCH3          ; ZCHAR_SCRATCH2[dest_index] = c >> 5
    LSR
    LSR
    LSR
    LSR
    LSR
    AND     #$03
    LDX     SCRATCH1
    STA     ZCHAR_SCRATCH2,X
    INC     SCRATCH1          ; ++dest_index
    DEC     SCRATCH3+1        ; --nchars
    BEQ     z_compress

    LDA     SCRATCH3          ; c &= 0x1F
    AND     #$1F
    JMP     .store_zchar
```

Uses SCRATCH1 204, SCRATCH3 204, ZCHAR_SCRATCH2 204, and z_compress 85.

Searching the dictionary

The address of the dictionary is stored in the header, and the `get_dictionary_addr` routine gets the absolute address of the dictionary and stores it in `SCRATCH2`.

90 ⟨*Get dictionary address* 90⟩≡ (207)

```
get_dictionary_addr:
    SUBROUTINE

    LDY      #HEADER_DICT_OFFSET
    LDA      (Z_HEADER_ADDR),Y
    STA      SCRATCH2+1
    INY
    LDA      (Z_HEADER_ADDR),Y
    STA      SCRATCH2
    ADDW     SCRATCH2, Z_HEADER_ADDR, SCRATCH2
    RTS
```

Defines:
`get_dictionary_addr`, used in chunks 80 and 91.

Uses `ADDW` 15c, `HEADER_DICT_OFFSET` 206a, and `SCRATCH2` 204.

The match_dictionary_word routines searches for a word in the dictionary, returning in SCRATCH1 the z-address of the matching dictionary entry, or zero if not found.

91 ⟨*Match dictionary word* 91⟩≡ (207) 92a ▷

```
match_dictionary_word:
    SUBROUTINE

    JSR     get_dictionary_addr
    LDY     #$00                ; number of dict separators
    LDA     (SCRATCH2),Y
    TAY                         ; skip past and get entry length
    INY
    LDA     (SCRATCH2),Y
    ASL                         ; search_size = entry length x 16
    ASL
    ASL
    ASL
    STA     SCRATCH3
    INY                         ; entry_index = num dict entries
    LDA     (SCRATCH2),Y
    STA     SCRATCH1+1
    INY
    LDA     (SCRATCH2),Y
    STA     SCRATCH1
    INY
    TYA
    ADDA    SCRATCH2            ; entry_addr = start of dictionary entries
    LDY     #$00
    JMP     .try_match
```

Defines:
match_dictionary_word, used in chunk 78.

Uses ADDA 14a, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, and get_dictionary_addr 90.

Since the dictionary is stored in lexicographic order, if we ever find a word that is greater than the word we are looking for, or we reach the end of the dictionary, then we can stop searching.

Instead of searching incrementally, we actually search in steps of 16 entries. When we've located the chunk of entries that our word should be in, we then search through the 16 entries to find the word, or fail.

92a ⟨*Match dictionary word* 91⟩+≡ (207) ◁91 92b▷

```
.loop:
    LDA      (SCRATCH2),Y
    CMP      ZCHAR_SCRATCH2+1
    BCS      .possible

.try_match:
    ADDB2    SCRATCH2, SCRATCH3      ; entry_addr += search_size
    SEC                              ; entry_index -= 16
    LDA      SCRATCH1
    SBC      #$10
    STA      SCRATCH1
    BCS      .loop
    DEC      SCRATCH1+1
    BPL      .loop
```

Uses ADDB2 15b, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, and ZCHAR_SCRATCH2 204.

92b ⟨*Match dictionary word* 91⟩+≡ (207) ◁92a 93▷

```
.possible:
    SUBB2    SCRATCH2, SCRATCH3      ; entry_addr -= search_size
    ADDB2    SCRATCH1, #$10          ; entry_index += 16
    LDA      SCRATCH3                ; search_size /= 16
    LSR
    LSR
    LSR
    LSR
    STA      SCRATCH3
```

Uses ADDB2 15b, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, and SUBB2 17a.

Now we compare the word. The words in the dictionary are numerically big-endian while the words in the ZCHAR_SCRATCH2 buffer are numerically little-endian, which explains the unusual order of the comparisons.

Since we know that the dictionary word must be in this chunk of 16 words if it exists, then if our word is less than the dictionary word, we can stop searching and declare failure.

93 ⟨*Match dictionary word* 91⟩+≡ (207) ◁92b 94a▷

```
.inner_loop:
    LDY      #$00
    LDA      ZCHAR_SCRATCH2+1
    CMP      (SCRATCH2),Y
    BCC      .not_found
    BNE      .inner_next

    INY
    LDA      ZCHAR_SCRATCH2
    CMP      (SCRATCH2),Y
    BCC      .not_found
    BNE      .inner_next

    LDY      #$02
    LDA      ZCHAR_SCRATCH2+3
    CMP      (SCRATCH2),Y
    BCC      .not_found
    BNE      .inner_next

    INY
    LDA      ZCHAR_SCRATCH2+2
    CMP      (SCRATCH2),Y
    BCC      .not_found
    BEQ      .found

.inner_next:
    ADDB2    SCRATCH2, SCRATCH3     ; entry_addr += search_size
    SUBB     SCRATCH1, #$01         ; --entry_index
    LDA      SCRATCH1
    ORA      SCRATCH1+1
    BNE      .inner_loop
```

Uses ADDB2 15b, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, SUBB 16b, and ZCHAR_SCRATCH2 204.

If the search failed, we return 0 in SCRATCH1.

94a ⟨*Match dictionary word* 91⟩+≡ (207) ◁93 94b▷

```
.not_found:
    LDA     #$00
    STA     SCRATCH1+1
    STA     SCRATCH1
    RTS
```

Uses SCRATCH1 204.

Otherwise, return the z-address (i.e. the absolute address minus the header address) of the dictionary entry.

94b ⟨*Match dictionary word* 91⟩+≡ (207) ◁94a

```
.found:
    SUBW    SCRATCH2, Z_HEADER_ADDR, SCRATCH1
    RTS
```

Uses SCRATCH1 204, SCRATCH2 204, and SUBW 17b.

Chapter 8

Arithmetic routines

8.0.1 Negation and sign manipulation

`negate` negates the word in `SCRATCH2`.

95a ⟨*negate* 95a⟩≡ (207)

```
negate:
    SUBROUTINE

    SUBW    #$0000, SCRATCH2, SCRATCH2
    RTS
```

Defines:
`negate`, used in chunks 95b, 96b, and 104.
Uses `SCRATCH2` 204 and `SUBW` 17b.

`flip_sign` negates the word in `SCRATCH2` if the sign bit in the `A` register is set, i.e. if signed `A` is negative. We also keep track of the number of flips in `SIGN_BIT`.

95b ⟨*Flip sign* 95b⟩≡ (207)

```
flip_sign:
    SUBROUTINE

    ORA     #$00
    BMI     .do_negate
    RTS

.do_negate:
    INC     SIGN_BIT
    JMP     negate
```

Defines:
`flip_sign`, used in chunk 96a.
Uses `negate` 95a.

check_sign sets the sign bit of SCRATCH2 to support a 16-bit signed multiply, divide, or modulus operation on SCRATCH1 and SCRATCH2. That is, if the sign bits are the same, SCRATCH2 retains its sign bit, otherwise its sign bit is flipped.

The SIGN_BIT value also contains the number of negative sign bits in SCRATCH1 and SCRATCH2, so 0, 1, or 2.

96a ⟨*Check sign* 96a⟩≡ (207)

```
check_sign:
    SUBROUTINE

    LDA       #$00
    STA       SIGN_BIT
    LDA       SCRATCH2+1
    JSR       flip_sign
    LDA       SCRATCH1+1
    JSR       flip_sign
    RTS
```

Defines:
check_sign, used in chunks 171–73.
Uses SCRATCH1 204, SCRATCH2 204, and flip_sign 95b.

set_sign checks the number of negatives counted up in SIGN_BIT and sets the sign bit of SCRATCH2 accordingly. That is, odd numbers of negative signs will flip the sign bit of SCRATCH2.

96b ⟨*Set sign* 96b⟩≡ (207)

```
set_sign:
    SUBROUTINE

    LDA       SIGN_BIT
    AND       #$01
    BNE       negate
    RTS
```

Defines:
set_sign, used in chunk 173.
Uses negate 95a.

8.0.2 16-bit multiplication

`mulu16` multiples the unsigned word in `SCRATCH1` by the unsigned word in `SCRATCH2`, storing the result in `SCRATCH1`.

Note that this routine only handles unsigned multiplication. Taking care of signs is part of `instr_mul`, which uses this routine and the sign manipulation routines.

97 ⟨*mulu16* 97⟩≡ (207)

```
mulu16:
    SUBROUTINE

    PSHW     SCRATCH3
    STOW     #$0000, SCRATCH3
    LDX      #$10

.loop:
    LDA      SCRATCH1
    CLC
    AND      #$01
    BEQ      .next_bit
    ADDWC    SCRATCH2, SCRATCH3, SCRATCH3

.next_bit:
    RORW      SCRATCH3
    RORW      SCRATCH1
    DEX
    BNE      .loop

    MOVW     SCRATCH1, SCRATCH2
    MOVW     SCRATCH3, SCRATCH1
    PULW     SCRATCH3
    RTS
```

Defines:
`mulu16`, used in chunk 173.

Uses `ADDWC` 16a, `MOVW` 12a, `PSHW` 12b, `PULW` 13a, `RORW` 18, `SCRATCH1` 204, `SCRATCH2` 204, `SCRATCH3` 204, and `STOW` 10.

8.0.3 16-bit division

`divu16` divides the unsigned word in `SCRATCH2` (the dividend) by the unsigned word in `SCRATCH1` (the divisor), storing the quotient in `SCRATCH2` and the remainder in `SCRATCH1`.

Under this routine, the result of division by zero is a quotient of $2^{16} - 1$, while the remainder depends on the high bit of the dividend. If the dividend's high bit is 0, the remainder is the dividend. If the dividend's high bit is 1, the remainder is the dividend with the high bit set to 0.

Note that this routine only handles unsigned division. Taking care of signs is part of `instr_div`, which uses this routine and the sign manipulation routines.

The idea behind this routine is to do long division. We bring the dividend into a scratch space one bit at a time (starting with the most significant bit) and see if the divisor fits into it. It it does, we can record a 1 in the quotient, and subtract the divisor from the scratch space. If it doesn't, we record a 0 in the quotient. We do this for all 16 bits in the dividend. Whatever remains in the scratch space is the remainder.

For example, suppose we want to divide decimal `SCRATCH2 = 37 = 0b10101` by `SCRATCH1 = 10 = 0b1010`. This is something the `print_number` routine might do.

The routine starts with storing `SCRATCH2` to `SCRATCH3 = 37 = 0b100101` and then setting `SCRATCH2` to zero. This is our scratch space, and will ultimately become the remainder.

Interestingly here, we don't start with shifting the dividend. Instead we do the subtraction first. There's no harm in this, since we are guaranteed that the subtraction will fail (be negative) on the first iteration, so we shift in a zero.

It should be clear that as we shift the dividend into the scratch space, eventually the scratch space will contain `0b10010`, and the subtraction will succeed. We then shift in a 1 into the quotient, and subtract the divisor `0b1010` from the scratch space `0b10010`, leaving `0b1000`. There is now only one bit left in the dividend (`1`).

We shift that into the scratch space, which is now `0b10001`, and the subtraction will succeed again. We shift in a 1 into the quotient, and subtract the divisor from the scratch space, leaving `0b111`. There are no bits left in the dividend, so we are done. The quotient is `0b11 = 3` and the scratch space is `0b111 = 7`, which is the remainder as expected.

Because the algorithm always does the shift, it will also shift the remainder one time too many, which is why the last step is to shift it right and store the result.

Here's a trace of the algorithm:

99 ⟨*trace of divu16* 99⟩≡

```
Begin, x=17: s1=0000000000001010, s2=0000000000000000, s3=0000000000100101
Loop,  x=16: s1=0000000000001010, s2=0000000000000000, s3=0000000001001010
Loop,  x=15: s1=0000000000001010, s2=0000000000000000, s3=0000000010010100
Loop,  x=14: s1=0000000000001010, s2=0000000000000000, s3=0000000100101000
Loop,  x=13: s1=0000000000001010, s2=0000000000000000, s3=0000001001010000
Loop,  x=12: s1=0000000000001010, s2=0000000000000000, s3=0000010010100000
Loop,  x=11: s1=0000000000001010, s2=0000000000000000, s3=0000100101000000
Loop,  x=10: s1=0000000000001010, s2=0000000000000000, s3=0001001010000000
Loop,  x=09: s1=0000000000001010, s2=0000000000000000, s3=0010010100000000
Loop,  x=08: s1=0000000000001010, s2=0000000000000000, s3=0100101000000000
Loop,  x=07: s1=0000000000001010, s2=0000000000000000, s3=1001010000000000
Loop,  x=06: s1=0000000000001010, s2=0000000000000001, s3=0010100000000000
Loop,  x=05: s1=0000000000001010, s2=0000000000000010, s3=0101000000000000
Loop,  x=04: s1=0000000000001010, s2=0000000000000100, s3=1010000000000000
Loop,  x=03: s1=0000000000001010, s2=0000000000001001, s3=0100000000000000
Loop,  x=02: s1=0000000000001010, s2=0000000000010010, s3=1000000000000000
Loop,  x=01: s1=0000000000001010, s2=0000000000010001, s3=0000000000000001
Loop,  x=00: s1=0000000000001010, s2=0000000000001110, s3=0000000000000011
End,   x=00: s1=0000000000001010, s2=0000000000001110, s3=0000000000000011
After adjustment shift and remainder storage:
End,   x=00: s1=0000000000000111, s2=0000000000000011
```

Notice that SCRATCH3 is used for both the dividend and the quotient. As we shift bits out of the left of the dividend and into the scratch space SCRATCH2, we also shift bits into the right as the quotient. After going through 16 bits, the dividend is all out and the quotient is all in.

100 ⟨*divu16* 100⟩≡ (207)

```
divu16:
    SUBROUTINE

    PSHW     SCRATCH3
    MOVW     SCRATCH2, SCRATCH3 ; SCRATCH3 is the dividend
    STOW     #$0000, SCRATCH2   ; SCRATCH2 is the remainder
    LDX      #$11

.loop:
    SEC                         ; carry = "not borrow"
    LDA      SCRATCH2           ; Remainder minus divisor (low byte)
    SBC      SCRATCH1
    TAY
    LDA      SCRATCH2+1
    SBC      SCRATCH1+1
    BCC      .skip              ; Divisor did not fit

    ; At this point carry is set, which will affect
    ; the ROLs below.

    STA      SCRATCH2+1         ; Save remainder
    TYA
    STA      SCRATCH2

.skip:
    ROLW     SCRATCH3           ; Shift carry into divisor/quotient left
    ROLW     SCRATCH2           ; Shift divisor/remainder left
    DEX
    BNE      .loop              ; loop end

    CLC                         ; SCRATCH1 = SCRATCH2 >> 1
    LDA      SCRATCH2+1
    ROR
    STA      SCRATCH1+1
    LDA      SCRATCH2
    ROR
    STA      SCRATCH1               ; remainder
    MOVW     SCRATCH3, SCRATCH2 ; quotient
    PULW     SCRATCH3
    RTS
```

Defines:
divu16, used in chunks 103, 171, 172, and 174a.

Uses MOVW 12a, PSHW 12b, PULW 13a, ROLW 17c, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, and STOW 10.

8.0.4 16-bit comparison

`cmpu16` compares the unsigned words in `SCRATCH2` to the unsigned word in `SCRATCH1`. For example, if, as an unsigned comparison, `SCRATCH2<SCRATCH1`, then `BCC` will detect this condition.

101a ⟨*cmpu16* 101a⟩≡ (207)

```
cmpu16:
    SUBROUTINE

    LDA     SCRATCH2+1
    CMP     SCRATCH1+1
    BNE     .end
    LDA     SCRATCH2
    CMP     SCRATCH1
.end:
    RTS
```

Defines:
`cmpu16`, used in chunks 101b and 181a.
Uses `SCRATCH1` 204 and `SCRATCH2` 204.

`cmp16` compares the two signed words in `SCRATCH1` and `SCRATCH2`.

101b ⟨*cmp16* 101b⟩≡ (207)

```
cmp16:
    SUBROUTINE

    LDA     SCRATCH1+1
    EOR     SCRATCH2+1
    BPL     cmpu16
    LDA     SCRATCH1+1
    CMP     SCRATCH2+1
    RTS
```

Defines:
`cmp16`, used in chunks 177a, 179a, and 180a.
Uses `SCRATCH1` 204, `SCRATCH2` 204, and `cmpu16` 101a.

8.0.5 Other routines

`A_mod_3` is a routine that calculates the modulus of the `A` register with 3, by repeatedly subtracting 3 until the result is less than 3. ¡3 It is used in the Z-machine to calculate the alphabet shift.

102 ⟨*A mod 3* 102⟩≡ (207)

```
A_mod_3:
    CMP     #$03
    BCC     .end
    SEC
    SBC     #$03
    JMP     A_mod_3

.end:
    RTS
```

Defines:
`A_mod_3`, used in chunks 64, 84a, and 86.

8.0.6 Printing numbers

The print_number routine prints the signed number in SCRATCH2 as decimal to the output buffer.

103 ⟨*Print number* 103⟩≡ (207)

```
print_number:
    SUBROUTINE

    LDA      SCRATCH2+1
    BPL      .print_positive
    JSR      print_negative_num

.print_positive:
    STOB     #$00, SCRATCH3

.loop:
    LDA      SCRATCH2+1
    ORA      SCRATCH2
    BEQ      .is_zero
    STOW     #$000A, SCRATCH1
    JSR      divu16
    LDA      SCRATCH1
    PHA
    INC      SCRATCH3
    JMP      .loop

.is_zero:
    LDA      SCRATCH3
    BEQ      .print_0

.print_digit:
    PLA
    CLC
    ADC      #$30            ; '0'
    JSR      buffer_char
    DEC      SCRATCH3
    BNE      .print_digit
    RTS

.print_0:
    LDA      #$30            ; '0'
    JMP      buffer_char
```

Defines:
print_number, used in chunks 69 and 186a.

Uses SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, STOB 11b, STOW 10, buffer_char 57, divu16 100, and print_negative_num 104.

The `print_negative_num` routine is a utility used by `print_num`, just to print the negative sign and negate the number before printing the rest.

104 ⟨*Print negative number* 104⟩≡ (207)

```
print_negative_num:
    SUBROUTINE

    LDA     #$2D            ; '-'
    JSR     buffer_char
    JMP     negate
```

Defines:
`print_negative_num`, used in chunk 103.

Uses `buffer_char` 57 and `negate` 95a.

Chapter 9

Disk routines

105 ⟨*iob struct* 105⟩≡ (207)

```
iob:
    DC      #$01            ; table_type (must be 1)
iob.slot_times_16:
    DC      #$60            ; slot_times_16
iob.drive:
    DC      #$01            ; drive_number
    DC      #$00            ; volume
iob.track:
    DC      #$00            ; track
iob.sector:
    DC      #$00            ; sector
    DC.W    #dct            ; dct_addr
iob.buffer:
    DC.W    #$0000          ; buffer_addr
    DC      #$00            ; unused
    DC      #$00            ; partial_byte_count
iob.command:
    DC      #$00            ; command
    DC      #$00            ; ret_code
    DC      #$00            ; last_volume
    DC      #$60            ; last_slot_times_16
    DC      #$00            ; last_drive_number

dct:
    DC      #$00            ; device_type (0 for DISK II)
    DC      #$01            ; phases_per_track (1 for DISK II)
dct.motor_count:
    DC.W    #$EFD8          ; motor_on_time_count ($EFD8 for DISK II)
```

Defines:
`dct`, used in chunk 108.
`iob`, used in chunks 106, 146, and 148.

`iob.buffer`, never used.
`iob.command`, never used.
`iob.drive`, never used.
`iob.sector`, never used.
`iob.slot_times_16`, never used.
`iob.track`, never used.

The `do_rwts_on_sector` can read or write a sector using the `RWTS` routine in `DOS`. `SCRATCH1` contains the sector number relative to track 3 sector 0 (and can be `>=16`), and `SCRATCH2` contains the buffer to read into or write from.

The `A` register contains the command: 1 for read, and 2 for write.

106 ⟨*Do RWTS on sector* 106⟩≡ (207)

```
do_rwts_on_sector:
    SUBROUTINE

    STA     iob.command
    LDA     SCRATCH2
    STA     iob.buffer
    LDA     SCRATCH2+1
    STA     iob.buffer+1
    LDA     #$03
    STA     iob.track
    LDA     SCRATCH1
    LDX     SCRATCH1+1
    SEC

.adjust_track:
    SBC     SECTORS_PER_TRACK
    BCS     .inc_track
    DEX
    BMI     .do_read
    SEC

.inc_track:
    INC     iob.track
    JMP     .adjust_track

.do_read:
    CLC
    ADC     SECTORS_PER_TRACK
    STA     iob.sector
    LDA     #$1D
    LDY     #$AC
    JSR     RWTS
    RTS
```

Defines:
`do_rwts_on_sector`, used in chunks 107 and 108.
Uses `RWTS` 204, `SCRATCH1` 204, `SCRATCH2` 204, `SECTORS_PER_TRACK` 204, and `iob` 105.

The `read_from_sector` routine reads the sector number in `SCRATCH1` from the disk into the buffer in `SCRATCH2`. Other entry points are `read_next_sector`, which sets the buffer to `BUFF_AREA`, increments `SCRATCH1` and then reads, and `inc_sector_and_read`, which does the same but assumes the buffer has already been set in `SCRATCH2`.

107 ⟨*Reading sectors* 107⟩≡ (207)

```
read_next_sector:
    SUBROUTINE

    STOW     #BUFF_AREA, SCRATCH2

inc_sector_and_read:
    SUBROUTINE

    INCW     SCRATCH1

read_from_sector:
    SUBROUTINE

    LDA      #$01
    JSR      do_rwts_on_sector
    RTS
```

Defines:
- `inc_sector_and_read`, used in chunk 154b.
- `read_from_sector`, used in chunks 29b, 30, 40, and 43.
- `read_next_sector`, used in chunks 152c and 154a.

Uses `BUFF_AREA` 204, `INCW` 13b, `SCRATCH1` 204, `SCRATCH2` 204, `STOW` 10, and `do_rwts_on_sector` 106.

For some reason, possibly a bug, possibly an ill-informed optimization, the `write_next_sector` routine temporarily stores `#$D8EF` into the disk motor on-time count, where normally this is `#$EFD8`. There doesn't seem to be any reason for this, since the motor count is never set to anything else.

108 ⟨*Writing sectors* 108⟩≡ (207)

```
write_next_sector:
    SUBROUTINE

    STOW     #BUFF_AREA, SCRATCH2

inc_sector_and_write:
    SUBROUTINE

    INCW     SCRATCH1

.write_next_sector:
    LDA      dct.motor_count
    PHA
    LDA      dct.motor_count+1
    PHA
    STOW2    #$D8EF, dct.motor_count
    LDA      #$02
    JSR      do_rwts_on_sector
    PLA
    STA      dct.motor_count+1
    PLA
    STA      dct.motor_count
    RTS
```

Defines:
`inc_sector_and_write`, used in chunk 151b.
`write_next_sector`, used in chunks 150b and 151a.

Uses `BUFF_AREA` 204, `INCW` 13b, `SCRATCH1` 204, `SCRATCH2` 204, `STOW` 10, `STOW2` 11a, `dct` 105, and `do_rwts_on_sector` 106.

Chapter 10

The instruction dispatcher

10.1 Executing an instruction

The addresses for instructions handlers are stored in tables, organized by number of operands:

109 ⟨*Instruction tables* 109⟩≡ (207)

```
routines_table_0op:
    WORD    instr_rtrue
    WORD    instr_rfalse
    WORD    instr_print
    WORD    instr_print_ret
    WORD    instr_nop
    WORD    instr_save
    WORD    instr_restore
    WORD    instr_restart
    WORD    instr_ret_popped
    WORD    instr_pop
    WORD    instr_quit
    WORD    instr_new_line

routines_table_1op:
    WORD    instr_jz
    WORD    instr_get_sibling
    WORD    instr_get_child
    WORD    instr_get_parent
    WORD    instr_get_prop_len
    WORD    instr_inc
    WORD    instr_dec
    WORD    instr_print_addr
    WORD    illegal_opcode
```

```
      WORD    instr_remove_obj
      WORD    instr_print_obj
      WORD    instr_ret
      WORD    instr_jump
      WORD    instr_print_paddr
      WORD    instr_load
      WORD    instr_not

  routines_table_2op:
      WORD    illegal_opcode
      WORD    instr_je
      WORD    instr_jl
      WORD    instr_jg
      WORD    instr_dec_chk
      WORD    instr_inc_chk
      WORD    instr_jin
      WORD    instr_test
      WORD    instr_or
      WORD    instr_and
      WORD    instr_test_attr
      WORD    instr_set_attr
      WORD    instr_clear_attr
      WORD    instr_store
      WORD    instr_insert_obj
      WORD    instr_loadw
      WORD    instr_loadb
      WORD    instr_get_prop
      WORD    instr_get_prop_addr
      WORD    instr_get_next_prop
      WORD    instr_add
      WORD    instr_sub
      WORD    instr_mul
      WORD    instr_div
      WORD    instr_mod

  routines_table_var:
      WORD    instr_call
      WORD    instr_storew
      WORD    instr_storeb
      WORD    instr_put_prop
      WORD    instr_sread
      WORD    instr_print_char
      WORD    instr_print_num
      WORD    instr_random
      WORD    instr_push
      WORD    instr_pull
```

Defines:
`routines_table_0op`, used in chunk 113b.
`routines_table_1op`, used in chunk 115b.
`routines_table_2op`, used in chunk 117c.

routines_table_var, used in chunk 119.
Uses illegal_opcode 158, instr_add 170b, instr_and 175a, instr_call 125, instr_clear_attr 187, instr_dec 170a, instr_dec_chk 176b, instr_div 171, instr_get_next_prop 189, instr_get_parent 190, instr_get_prop 191, instr_get_prop_addr 194, instr_get_prop_len 195, instr_get_sibling 196, instr_inc 169c, instr_inc_chk 177a, instr_insert_obj 197, instr_je 177b, instr_jg 179a, instr_jin 179b, instr_jl 180a, instr_jump 182a, instr_jz 180b, instr_load 165a, instr_loadb 166a, instr_loadw 165b, instr_mod 172, instr_mul 173, instr_new_line 184b, instr_nop 200a, instr_not 175b, instr_or 176a, instr_pop 168b, instr_print 185a, instr_print_addr 185b, instr_print_char 185c, instr_print_num 186a, instr_print_obj 186b, instr_print_paddr 186c, instr_print_ret 182b, instr_pull 169a, instr_push 169b, instr_put_prop 198, instr_quit 201, instr_random 174a, instr_remove_obj 199a, instr_restart 200b, instr_restore 152b, instr_ret 129, instr_ret_popped 183a, instr_rfalse 183b, instr_rtrue 184a, instr_save 149a, instr_set_attr 199b, instr_sread 73, instr_store 166b, instr_storeb 168a, instr_storew 167, instr_sub 174c, instr_test 181a, and instr_test_attr 181b.

Instructions from this table get executed with all operands loaded in OPERAND0-OPERAND3, the address of the routine table to use in SCRATCH2, and the index into the table stored in the A register. Then we can execute the instruction. This involves looking up the routine address, storing it in SCRATCH1, and jumping to it.

All instructions must, when they are complete, jump back to do_instruction.

111 ⟨*Execute instruction* 111⟩≡ (207)

```
.opcode_table_jump:
    ASL
    TAY
    LDA     (SCRATCH2),Y
    STA     SCRATCH1
    INY
    LDA     (SCRATCH2),Y
    STA     SCRATCH1+1
    JSR     DEBUG_JUMP
    JMP     (SCRATCH1)
```

Defines:
.opcode_table_jump, never used.
Uses DEBUG_JUMP 204, SCRATCH1 204, and SCRATCH2 204.

The call to debug is just a return, but I suspect that it was used during development to provide a place to put a debugging hook, for example, to print out the state of the Z-machine on every instruction.

10.2 Retrieving the instruction

We execute the instruction at the current program counter by first retrieving its opcode. get_next_code_byte retrieves the code byte at Z_PC, placing it in A, and then increments Z_PC.

112 ⟨*Do instruction* 112⟩≡ (207) 113a ▷

```
do_instruction:
    SUBROUTINE

    MOVW    Z_PC, TMP_Z_PC      ; Save PC for debugging
    LDA     Z_PC+2
    STA     TMP_Z_PC+2
    STOB    #$00, OPERAND_COUNT
    JSR     get_next_code_byte
    STA     CURR_OPCODE
```

Defines:
do_instruction, used in chunks 33b, 74, 128b, 159, 161b, 164, 166–70, 184–87, and 197–200.

Uses CURR_OPCODE 204, MOVW 12a, OPERAND_COUNT 204, STOB 11b, TMP_Z_PC 204, Z_PC 204, and get_next_code_byte 37.

Byte range	Type
0x00-0x7F	2op
0x80-0xAF	1op
0xB0-0xBF	0op
0xC0-0xFF	needs next byte to determine

10.3 Decoding the instruction

Next, we determine how many operands to read. Note that for instructions that store a value, the storage location is not part of the operands; it comes after the operands, and is determined by the individual instruction's routine.

113a ⟨*Do instruction* 112⟩+≡ (207) ◁112

```
    CMP     #$80            ; is 2op?
    BCS     .is_gte_80
    JMP     .do_2op

.is_gte_80:
    CMP     #$B0            ; is 1op?
    BCS     .is_gte_B0
    JMP     .do_1op

.is_gte_B0:
    CMP     #$C0            ; is 0op?
    BCC     .do_0op
    JSR     get_next_code_byte

    ; Falls through to varop handling.

⟨Handle varop instructions 118⟩
```

Uses `get_next_code_byte` 37.

10.3.1 0op instructions

Handling a 0op-type instruction is easy enough. We check for the legal opcode range (`#$B0`-`#$BB`), otherwise it's an illegal instruction. Then we load the address of the 0op instruction table into `SCRATCH2`, leaving the `A` register with the offset into the table of the instruction to execute.

113b ⟨*Handle 0op instructions* 113b⟩≡ (207)

```
.do_0op:
    SEC
    SBC     #$B0
    CMP     #$0C
    BCC     .load_opcode_table
    JMP     illegal_opcode

.load_opcode_table:
    PHA
    STOW    routines_table_0op, SCRATCH2
    PLA
    JMP     .opcode_table_jump
```

Uses `SCRATCH2` 204, `STOW` 10, `illegal_opcode` 158, and `routines_table_0op` 109.

10.3.2 1op instructions

Handling a 1op-type instruction (opcodes `#$80-#$AF`) is a little more complicated. Since only opcodes `#$X8` are illegal, this is handled in the 1op routine table.

Opcodes `#$80-#$8F` take a 16-bit operand.

114a ⟨*Handle 1op instructions* 114a⟩≡ (207) 114b▷

```
.do_1op:
    AND     #$30
    BNE     .is_90_to_AF
    JSR     get_const_word   ; Get operand for opcodes 80-8F
    JMP     .1op_arg_loaded
```

Uses `get_const_word` 120b.

Opcodes `#$90-#$9F` take an 8-bit operand zero-extended to 16 bits.

114b ⟨*Handle 1op instructions* 114a⟩+≡ (207) ◁114a 114c▷

```
.is_90_to_AF:
    CMP     #$10
    BNE     .is_A0_to_AF
    JSR     get_const_byte   ; Get operand for opcodes 90-9F
    JMP     .1op_arg_loaded
```

Uses `get_const_byte` 120a.

Opcodes `#$A0-#$AF` take a variable number operand, whose content is 16 bits.

114c ⟨*Handle 1op instructions* 114a⟩+≡ (207) ◁114b 114d▷

```
.is_A0_to_AF:
    JSR     get_var_content  ; Get operand for opcodes A0-AF
```

Uses `get_var_content` 121.

The resulting 16-bit operand is placed in `OPERAND0`, and `OPERAND_COUNT` is set to 1.

114d ⟨*Handle 1op instructions* 114a⟩+≡ (207) ◁114c 115a▷

```
.1op_arg_loaded:
    STOB    #$01, OPERAND_COUNT
    MOVW    SCRATCH2, OPERAND0
```

Uses `MOVW` 12a, `OPERAND0` 204, `OPERAND_COUNT` 204, `SCRATCH2` 204, and `STOB` 11b.

Then we check for illegal instructions, which in this case never happens. This could have been left over from a previous version of the z-machine where the range of legal 1op instructions was different.

115a ⟨*Handle 1op instructions* 114a⟩+≡ (207) ◁114d 115b▷

```
        LDA     CURR_OPCODE
        AND     #$0F
        CMP     #$10
        BCC     .go_to_1op
        JMP     illegal_opcode
```

Uses CURR_OPCODE 204 and illegal_opcode 158.

Then we load the 1op instruction table into SCRATCH2, leaving the A register with the offset into the table of the instruction to execute.

115b ⟨*Handle 1op instructions* 114a⟩+≡ (207) ◁115a

```
    .go_to_1op:
        PHA
        STOW    routines_table_1op, SCRATCH2
        PLA
        JMP     .opcode_table_jump
```

Uses SCRATCH2 204, STOW 10, and routines_table_1op 109.

10.3.3 2op instructions

Handling a 2op-type instruction (opcodes `#$00-#$7F`) is a little more complicated than 1op instructions.

The operands are determined by bits 6 and 5, while bits 4 through 0 determine the instruction.

The first operand is determined by bit 6. Opcodes with bit 6 clear are followed by a single byte to be zero-extended into a 16-bit operand, while opcodes with bit 6 set are followed by a single byte representing a variable number. This operand is stored in `OPERAND0`.

116a ⟨*Handle 2op instructions* 116a⟩≡ (207) 116b ▷

```
.do_2op:
    AND     #$40
    BNE     .first_arg_is_var
    JSR     get_const_byte
    JMP     .get_next_arg

.first_arg_is_var:
    JSR     get_var_content

.get_next_arg:
    MOVW    SCRATCH2, OPERAND0
```

Uses `MOVW` 12a, `OPERAND0` 204, `SCRATCH2` 204, `get_const_byte` 120a, and `get_var_content` 121.

The second operand is determined by bit 5. Opcodes with bit 5 clear are followed by a single byte to be zero-extended into a 16-bit operand, while opcodes with bit 5 set are followed by a single byte representing a variable number. This operand is stored in `OPERAND1`.

116b ⟨*Handle 2op instructions* 116a⟩+≡ (207) ◁ 116a 117a ▷

```
    LDA     CURR_OPCODE
    AND     #$20
    BNE     .second_arg_is_var
    JSR     get_const_byte
    JMP     .store_second_arg

.second_arg_is_var:
    JSR     get_var_content

.store_second_arg:
    MOVW    SCRATCH2, OPERAND1
```

Uses `CURR_OPCODE` 204, `MOVW` 12a, `OPERAND1` 204, `SCRATCH2` 204, `get_const_byte` 120a, and `get_var_content` 121.

OPERAND_COUNT is set to 2.

117a ⟨*Handle 2op instructions* 116a⟩+≡ (207) ◁116b 117b▷
```
    STOB      #$02, OPERAND_COUNT
```
Uses OPERAND_COUNT 204 and STOB 11b.

Then we check for illegal instructions, which are those with the low 5 bits in the range #$19-#$1F.

117b ⟨*Handle 2op instructions* 116a⟩+≡ (207) ◁117a 117c▷
```
    LDA       CURR_OPCODE

.check_for_good_2op:
    AND       #$1F
    CMP       #$19
    BCC       .go_to_op2
    JMP       illegal_opcode
```
Defines:
.check_for_good_2op, never used.
Uses CURR_OPCODE 204 and illegal_opcode 158.

Then we load the 2op instruction table into SCRATCH2, leaving the A register with the offset into the table of the instruction to execute.

117c ⟨*Handle 2op instructions* 116a⟩+≡ (207) ◁117b
```
.go_to_op2:
    PHA
    STOW      routines_table_2op, SCRATCH2
    PLA
    JMP       .opcode_table_jump
```
Uses SCRATCH2 204, STOW 10, and routines_table_2op 109.

Bits	Type	Bytes in operand
00	Large constant (0x0000-0xFFFF)	2
01	Small constant (0x00-0xFF)	1
10	Variable address	1
11	None (ends operand list)	0

10.3.4 varop instructions

Handling a varop-type instruction (opcodes `#$C0-#$FF`) is the most complicated. Interestingly, opcodes `#$C0-#$DF` map to 2op instructions (in their lower 5 bits).

The next byte is a map that determines the next operands. We look at two consecutive bits, starting from the most significant. The operand types are encoded as follows:

The values of the operands are stored consecutively starting in location `OPERAND0`.

118 ⟨*Handle varop instructions* 118⟩≡ (113a) 119▷

```
    LDX     #$00                ; operand number

.get_next_operand:
    PHA                         ; save operand map
    TAY
    TXA
    PHA                         ; save operand number
    TYA
    AND     #$C0                ; check top 2 bits
    BNE     .is_01_10_11
    JSR     get_const_word             ; handle 00
    JMP     .store_operand

.is_01_10_11:
    CMP     #$80
    BNE     .is_01_11
    JSR     get_var_content            ; handle 10
    JMP     .store_operand

.is_01_11:
    CMP     #$40
    BNE     .is_11
    JSR     get_const_byte             ; handle 01
    JMP     .store_operand

.is_11:
    PLA
    PLA
```

```
    JMP      .handle_varoperand_opcode ; handle 11 (ends operand list)

.store_operand:
    PLA
    TAX
    LDA      SCRATCH2
    STA      OPERAND0,X
    LDA      SCRATCH2+1
    STA      OPERAND0,X
    INX
    INX
    INC      OPERAND_COUNT
    PLA                                ; shift operand map left 2 bits
    SEC
    ROL
    SEC
    ROL
    JMP      .get_next_operand
```

Uses OPERAND0 204, OPERAND_COUNT 204, SCRATCH2 204, get_const_byte 120a, get_const_word 120b, and get_var_content 121.

Then we load the varop instruction table into SCRATCH2, leaving the A register with the offset into the table of the instruction to execute. However, we also check for illegal opcodes. Since opcodes #$C0-#$DF map to 2op instructions in their lower 5 bits, we simply hook into the 2op routine to do the opcode check and table jump.

Opcodes #$EA-#$FF are illegal.

119 ⟨*Handle varop instructions* 118⟩+≡ (113a) ◃118

```
.handle_varoperand_opcode:
    STOW     routines_table_var, SCRATCH2
    LDA      CURR_OPCODE
    CMP      #$E0
    BCS      .is_vararg_instr
    JMP      .check_for_good_2op

.is_vararg_instr:
    SBC      #$E0             ; Allow only E0-E9.
    CMP      #$0A
    BCC      .opcode_table_jump
    JMP      illegal_opcode
```

Uses CURR_OPCODE 204, SCRATCH2 204, STOW 10, illegal_opcode 158, and routines_table_var 109.

10.4 Getting the instruction operands

The utility routine `get_const_byte` gets the next byte of Z-code and stores it as a zero-extended 16-bit word in `SCRATCH2`.

120a ⟨*Get const byte* 120a⟩≡ (207)

```
get_const_byte:
    SUBROUTINE

    JSR     get_next_code_byte
    STA     SCRATCH2
    LDA     #$00
    STA     SCRATCH2+1
    RTS
```

Defines:
`get_const_byte`, used in chunks 114b, 116, and 118.
Uses `SCRATCH2` 204 and `get_next_code_byte` 37.

The utility routine `get_const_word` gets the next two bytes of Z-code and stores them as a 16-bit word in `SCRATCH2`. The word is stored big-endian in Z-code. The code in the routine is a little inefficient, since it uses the stack to shuffle bytes around, rather than storing the bytes directly in the right order.

120b ⟨*Get const word* 120b⟩≡ (207)

```
get_const_word:
    SUBROUTINE

    JSR     get_next_code_byte
    PHA
    JSR     get_next_code_byte
    STA     SCRATCH2
    PLA
    STA     SCRATCH2+1
    RTS
```

Defines:
`get_const_word`, used in chunks 114a and 118.
Uses `SCRATCH2` 204 and `get_next_code_byte` 37.

The utility routine `get_var_content` gets the next byte of Z-code and interprets it as a Z-variable address, then retrieves the variable's 16-bit value and stores it in `SCRATCH2`.

Variable `00` always means the top of the Z-stack, and this will also pop the stack.

Variables `01-0F` are "locals", and stored as 2-byte big-endian numbers in the zero-page at `$9A-$B9` (the `LOCAL_ZVARS` area).

Variables `10-FF` are "globals", and are stored as 2-byte big-endian numbers in a location stored at `GLOBAL_ZVARS_ADDR`.

121 ⟨*Get var content* 121⟩≡ (207)

```
get_var_content:
    SUBROUTINE

    JSR     get_next_code_byte          ; A = get_next_code_byte<Z_PC>
    ORA     #$00                        ; if (!A) get_top_of_stack
    BEQ     get_top_of_stack

get_nonstack_var:
    SUBROUTINE

    CMP     #$10                        ; if (A < #$10) {
    BCS     .compute_global_var_index
    SEC                                 ;   SCRATCH2 = LOCAL_ZVARS[A - 1]
    SBC     #$01
    ASL
    TAX
    LDA     LOCAL_ZVARS,X
    STA     SCRATCH2+1
    INX
    LDA     LOCAL_ZVARS,X
    STA     SCRATCH2
    RTS                                 ;   return
                                        ; }

.compute_global_var_index:
    SEC                                 ; var_ptr = 2 * (A - #$10)
    SBC     #$10
    ASL
    STA     SCRATCH1
    LDA     #$00
    ROL
    STA     SCRATCH1+1

.get_global_var_addr:
    CLC                                 ; var_ptr += GLOBAL_ZVARS_ADDR
    LDA     GLOBAL_ZVARS_ADDR
```

```
    ADC     SCRATCH1
    STA     SCRATCH1
    LDA     GLOBAL_ZVARS_ADDR+1
    ADC     SCRATCH1+1
    STA     SCRATCH1+1

.get_global_var_value:
    LDY     #$00                    ; SCRATCH2 = *var_ptr
    LDA     (SCRATCH1),Y
    STA     SCRATCH2+1
    INY
    LDA     (SCRATCH1),Y
    STA     SCRATCH2
    RTS                             ; return

get_top_of_stack:
    SUBROUTINE

    JSR     pop                     ; SCRATCH2 = pop()
    RTS                             ; return
```

Defines:
get_nonstack_var, used in chunk 122.
get_top_of_stack, never used.
get_var_content, used in chunks 114c, 116, and 118.
Uses GLOBAL_ZVARS_ADDR 204, LOCAL_ZVARS 204, SCRATCH1 204, SCRATCH2 204, Z_PC 204, get_next_code_byte 37, and pop 36.

There's another utility routine `var_get` which does the same thing, except the variable address is already stored in the `A` register.

122 ⟨*Get var content in A* 122⟩≡ (207)

```
var_get:
    SUBROUTINE

    ORA     #$00
    BEQ     pop_push
    JMP     get_nonstack_var
```

Defines:
var_get, used in chunks 69, 160, and 165a.
Uses get_nonstack_var 121 and pop_push 124.

The routine store_var stores SCRATCH2 into the variable in the next code byte, while store_var2 stores SCRATCH2 into the variable in the A register. Since variable 0 is the stack, storing into variable 0 is equivalent to pushing onto the stack.

123 ⟨*Store var* 123⟩≡ (207)

```
store_var:
    SUBROUTINE

    LDA     SCRATCH2                ; A = get_next_code_byte()
    PHA
    LDA     SCRATCH2+1
    PHA
    JSR     get_next_code_byte
    TAX
    PLA
    STA     SCRATCH2+1
    PLA
    STA     SCRATCH2
    TXA

store_var2:
    SUBROUTINE

    ORA     #$00
    BNE     .nonstack
    JMP     push

.nonstack:
    CMP     #$10
    BCS     .global_var
    SEC
    SBC     #$01
    ASL
    TAX
    LDA     SCRATCH2+1
    STA     LOCAL_ZVARS,X
    INX
    LDA     SCRATCH2
    STA     LOCAL_ZVARS,X
    RTS

.global_var:
    SEC
    SBC     #$10
    ASL
    STA     SCRATCH1
    LDA     #$00
    ROL
    STA     SCRATCH1+1
```

```
    CLC
    LDA     GLOBAL_ZVARS_ADDR
    ADC     SCRATCH1
    STA     SCRATCH1
    LDA     GLOBAL_ZVARS_ADDR+1
    ADC     SCRATCH1+1
    STA     SCRATCH1+1
    LDY     #$00
    LDA     SCRATCH2+1
    STA     (SCRATCH1),Y
    INY
    LDA     SCRATCH2
    STA     (SCRATCH1),Y
    RTS
```

Defines:
store_var, used in chunks 159a and 188.
Uses GLOBAL_ZVARS_ADDR 204, LOCAL_ZVARS 204, SCRATCH1 204, SCRATCH2 204, get_next_code_byte 37, and push 35.

The var_put routine stores the value in SCRATCH2 into the variable in the A register. Note that if the variable is 0, then it replaces the top value on the stack.

124 ⟨*Store to var A* 124⟩≡ (207)

```
var_put:
    SUBROUTINE

    ORA     #$00
    BEQ     .pop_push
    JMP     store_var2

pop_push:
    JSR     pop
    JMP     push

.pop_push:
    LDA     SCRATCH2
    PHA
    LDA     SCRATCH2+1
    PHA
    JSR     pop
    PLA
    STA     SCRATCH2+1
    PLA
    STA     SCRATCH2
    JMP     push
```

Defines:
pop_push, used in chunk 122.
var_put, used in chunks 160a and 166b.
Uses SCRATCH2 204, pop 36, and push 35.

Chapter 11

Calls and returns

11.1 Call

The `call` instruction calls the routine at the packed address in operand 0. A call may have anywhere from 0 to 3 arguments, and a routine always has a return value. Note that calls to address 0 merely returns false (0).

The z-code byte after the operands gives the variable in which to store the return value from the call.

125 ⟨*Instruction call* 125⟩≡ (207) 126a ▷

```
instr_call:
    LDA     OPERAND0
    ORA     OPERAND0+1
    BNE     .push_frame
    STOW    #$0000, SCRATCH2
    JMP     store_and_next
```

Defines:
`instr_call`, used in chunk 109.
Uses `OPERAND0` 204, `SCRATCH2` 204, `STOW` 10, and `store_and_next` 159a.

Packed addresses are byte addresses divided by two.

The routine's arguments are stored in local variables (starting from variable 1). Such used local variables are saved before the call, and restored after the call.

As usual with calls, calls push a frame onto the stack, while returns pop a frame off the stack.

The frame consists of the frame's stack count, Z_PC, and the frame's stack pointer.

126a ⟨*Instruction call* 125⟩+≡ (207) ◁125 126b▷

```
  .push_frame:
      MOVB     FRAME_STACK_COUNT, SCRATCH2
      MOVB     Z_PC, SCRATCH2+1
      JSR      push
      MOVW     FRAME_Z_SP, SCRATCH2
      JSR      push
      MOVW     Z_PC+1, SCRATCH2
      JSR      push
      STOB     #$00, ZCODE_PAGE_VALID
```

Uses FRAME_STACK_COUNT 204, FRAME_Z_SP 204, MOVB 11b, MOVW 12a, SCRATCH2 204, STOB 11b, ZCODE_PAGE_VALID 204, Z_PC 204, and push 35.

Next, we unpack the call address and put it in Z_PC.

126b ⟨*Instruction call* 125⟩+≡ (207) ◁126a 126c▷

```
      LDA      OPERAND0
      ASL
      STA      Z_PC
      LDA      OPERAND0+1
      ROL
      STA      Z_PC+1
      LDA      #$00
      ROL
      STA      Z_PC+2
```

Uses OPERAND0 204 and Z_PC 204.

The first byte in a routine is the number of local variables (0-15). We now retrieve it (and save it for later).

126c ⟨*Instruction call* 125⟩+≡ (207) ◁126b 127▷

```
      JSR      get_next_code_byte     ; local_var_count = get_next_code_byte()
      PHA                             ; Save local_var_count
      ORA      #$00
      BEQ      .after_loop2
```

Uses get_next_code_byte 37.

Now we push and initialize the local variables. The next words in the routine are the initial values of the local variables.

127 ⟨*Instruction call* 125⟩+≡ (207) ◁126c 128a▷

```
    LDX     #$00                ; X = 0

.push_and_init_local_vars:
    PHA                         ; Save local_var_count
    LDA     LOCAL_ZVARS,X       ; Push LOCAL_ZVAR[X] onto the stack
    STA     SCRATCH2+1
    INX
    LDA     LOCAL_ZVARS,X
    STA     SCRATCH2
    DEX
    TXA
    PHA
    JSR     push

    JSR     get_next_code_byte  ; SCRATCH2 = next init val
    PHA
    JSR     get_next_code_byte
    STA     SCRATCH2
    PLA
    STA     SCRATCH2+1

    PLA                         ; Restore local_var_count
    TAX
    LDA     SCRATCH2+1          ; LOCAL_ZVARS[X] = SCRATCH2
    STA     LOCAL_ZVARS,X
    INX
    LDA     SCRATCH2
    STA     LOCAL_ZVARS,X
    INX                         ; Increment X
    PLA                         ; Decrement local_var_count
    SEC
    SBC     #$01
    BNE     .push_and_init_local_vars  ; Loop until no more vars
```

Uses LOCAL_ZVARS 204, SCRATCH2 204, get_next_code_byte 37, and push 35.

Next, we load the local variables with the call arguments.

128a ⟨*Instruction call* 125⟩+≡ (207) ◁127 128b▷

```
.after_loop2:
    LDA      OPERAND_COUNT          ; count = OPERAND_COUNT - 1
    STA      SCRATCH3
    DEC      SCRATCH3
    BEQ      .done_init_local_vars  ; if (!count) .done_init_local_vars

    STOB     #$00, SCRATCH1         ; operand = 0
    STOB     #$00, SCRATCH2         ; zvar = 0

.loop:
    LDX      SCRATCH1               ; LOCAL_ZVARS[zvar] = OPERAND0[operand]
    LDA      OPERAND0+1,X
    LDX      SCRATCH2
    STA      LOCAL_ZVARS,X
    INC      SCRATCH2
    LDX      SCRATCH1
    LDA      OPERAND0,X
    LDX      SCRATCH2
    STA      LOCAL_ZVARS,X
    INC      SCRATCH2               ; ++zvar
    INC      SCRATCH1               ; ++operand
    INC      SCRATCH1
    DEC      SCRATCH3               ; --count
    BNE      .loop                  ; if (count) .loop
```

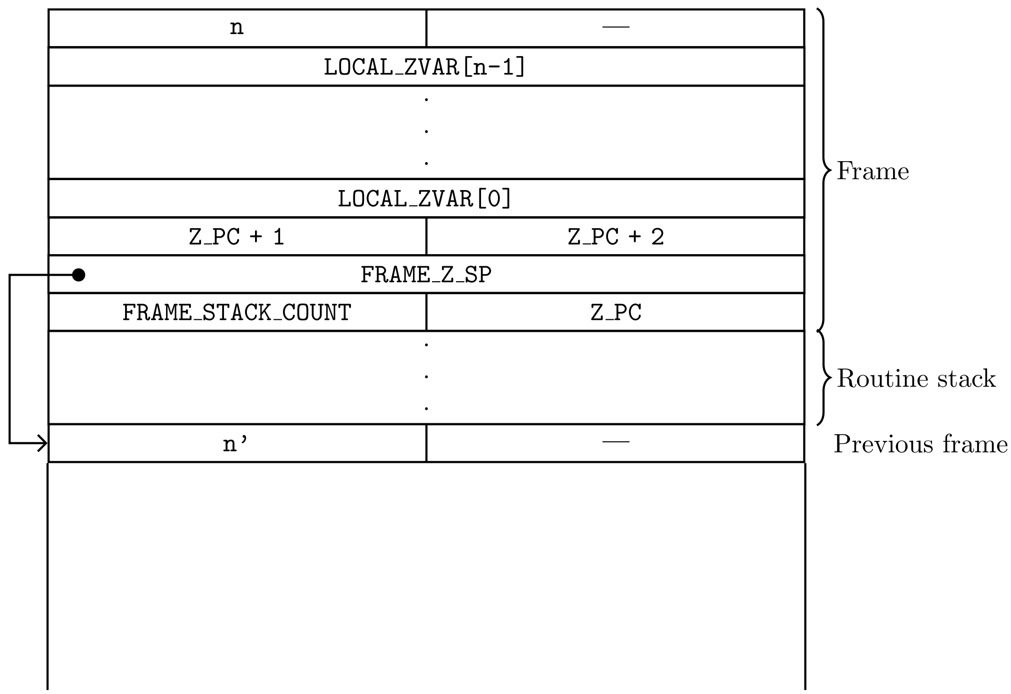
Uses LOCAL_ZVARS 204, OPERAND0 204, OPERAND_COUNT 204, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, and STOB 11b.

Finally, we add the local var count to the frame, update FRAME_STACK_COUNT and FRAME_Z_SP, and jump to the routine's first instruction.

128b ⟨*Instruction call* 125⟩+≡ (207) ◁128a

```
.done_init_local_vars:
    PULB     SCRATCH2               ; Restore local_var_count
    JSR      push                   ; Push local_var_count
    MOVB     STACK_COUNT, FRAME_STACK_COUNT
    MOVW     Z_SP, FRAME_Z_SP
    JMP      do_instruction
```

Uses FRAME_STACK_COUNT 204, FRAME_Z_SP 204, MOVB 11b, MOVW 12a, PULB 12c, SCRATCH2 204, STACK_COUNT 204, Z_SP 204, do_instruction 112, and push 35.



11.2 Return

The `ret` instruction returns from a routine. It effectively undoes what `call` did. First, we set the stack pointer and count to the frame's stack pointer and count.

129 ⟨*Instruction ret* 129⟩≡ (207) 130a ▷

```
instr_ret:
    SUBROUTINE

    MOVW    FRAME_Z_SP, Z_SP
    MOVB    FRAME_STACK_COUNT, STACK_COUNT
```

Defines:
`instr_ret`, used in chunks 109, 183a, and 184a.

Uses `FRAME_STACK_COUNT` 204, `FRAME_Z_SP` 204, `MOVB` 11b, `MOVW` 12a, `STACK_COUNT` 204, and `Z_SP` 204.

Next, we restore the locals. We first pop the number of locals off the stack, and if there were none, we can skip the whole local restore process.

130a ⟨*Instruction ret* 129⟩+≡ (207) ◁129 130b▷

```
    JSR     pop
    LDA     SCRATCH2
    BEQ     .done_locals
```

Uses SCRATCH2 204 and pop 36.

We then set up the loop variables for restoring the locals.

130b ⟨*Instruction ret* 129⟩+≡ (207) ◁130a 130c▷

```
    STOW    LOCAL_ZVARS-2, SCRATCH1     ; ptr = &LOCAL_ZVARS[-1]
    MOVB    SCRATCH2, SCRATCH3          ; count = STRATCH2
    ASL                             ; ptr += 2 * count
    ADDA    SCRATCH1
```

Uses ADDA 14a, LOCAL_ZVARS 204, MOVB 11b, SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, and STOW 10.

Now we pop the locals off the stack in reverse order.

130c ⟨*Instruction ret* 129⟩+≡ (207) ◁130b 130d▷

```
.loop:
    JSR     pop                 ; SCRATCH2 = pop()
    LDY     #$01                ; *ptr = SCRATCH2
    LDA     SCRATCH2
    STA     (SCRATCH1),Y
    DEY
    LDA     SCRATCH2+1
    STA     (SCRATCH1),Y
    SUBB    SCRATCH1, #$02      ; ptr -= 2
    DEC     SCRATCH3            ; --count
    BNE     .loop
```

Uses SCRATCH1 204, SCRATCH2 204, SCRATCH3 204, SUBB 16b, and pop 36.

Next, we restore Z_PC and the frame stack pointer and count.

130d ⟨*Instruction ret* 129⟩+≡ (207) ◁130c 131▷

```
.done_locals:
    JSR     pop
    MOVW    SCRATCH2, Z_PC+1
    JSR     pop
    MOVW    SCRATCH2, FRAME_Z_SP
    JSR     pop
    MOVB    SCRATCH2+1, Z_PC
    MOVB    SCRATCH2, FRAME_STACK_COUNT
```

Uses FRAME_STACK_COUNT 204, FRAME_Z_SP 204, MOVB 11b, MOVW 12a, SCRATCH2 204, Z_PC 204, and pop 36.

Finally, we store the return value.

131 ⟨*Instruction ret* 129⟩+≡ (207) ◁130d

```
    STOB    #$00, ZCODE_PAGE_VALID
    MOVW    OPERAND0, SCRATCH2
    JMP     store_and_next
```

Uses MOVW 12a, OPERAND0 204, SCRATCH2 204, STOB 11b, ZCODE_PAGE_VALID 204, and store_and_next 159a.

Chapter 12

Objects

12.1 Object table format

Objects are stored in an object table, and there are at most 255 of them. They are numbered from 1 to 255, and object 0 is the "nothing" object.

The object table contains 31 words (62 bytes) for property defaults, and then at most 255 objects, each containing 9 bytes.

The first 4 bytes of each object entry are 32 bits of attribute flags (offsets 0-3). Next is the parent object number (offset 4), the sibling object number (offset 5), and the child object number (offset 6). Finally, there are two bytes of properties (offsets 7 and 8).

12.2 Getting an object's address

The `get_object_address` routine gets the address of the object number in the `A` register and puts it in `SCRATCH2`.

It does this by first setting `SCRATCH2` to 9 times the `A` register (since objects entries are 9 bytes long).

132 ⟨*Get object address* 132⟩≡ (207) 133a ▷

```
get_object_addr:
    SUBROUTINE

    STA     SCRATCH2
    LDA     #$00
```

```
        STA     SCRATCH2+1
        LDA     SCRATCH2
        ASL     SCRATCH2
        ROL     SCRATCH2+1
        ASL     SCRATCH2
        ROL     SCRATCH2+1
        ASL     SCRATCH2
        ROL     SCRATCH2+1
        CLC
        ADC     SCRATCH2
        BCC     .continue
        INC     SCRATCH2+1
        CLC

    .continue:
```

Defines:
`get_object_addr`, used in chunks 134, 136–38, 140, 179b, 188, 190, 196, and 197.
Uses `SCRATCH2` 204.

Next, we add `FIRST_OBJECT_OFFSET` (53) to `SCRATCH2`. This skips the 31 words of property defaults, which would be 62 bytes, but since object numbers start from 1, the first object is at `53+9=62` bytes.

133a ⟨*Get object address* 132⟩+≡ (207) ◁132 133b▷

```
        ADC     #FIRST_OBJECT_OFFSET
        STA     SCRATCH2
        BCC     .continue2
        INC     SCRATCH2+1

    .continue2:
```

Uses `FIRST_OBJECT_OFFSET` 206a and `SCRATCH2` 204.

Finally, we get the address of the object table stored in the header and add it to `SCRATCH2`. The resulting address is thus in `SCRATCH2`.

133b ⟨*Get object address* 132⟩+≡ (207) ◁133a

```
        LDY     #HEADER_OBJECT_TABLE_ADDR_OFFSET-1
        LDA     (Z_HEADER_ADDR),Y
        CLC
        ADC     SCRATCH2
        STA     SCRATCH2
        DEY
        LDA     (Z_HEADER_ADDR),Y
        ADC     SCRATCH2+1
        ADC     Z_HEADER_ADDR+1
        STA     SCRATCH2+1
        RTS
```

Uses `HEADER_OBJECT_TABLE_ADDR_OFFSET` 206a and `SCRATCH2` 204.

12.3 Removing an object

The `remove_obj` routine removes the object number in `OPERAND0` from the object tree. This detaches the object from its parent, but the object retains its children.

Recall that an object is a node in a linked list. Each node contains a pointer to its parent, a pointer to its sibling (the next child of the parent), and a pointer to its first child. The null pointer is zero.

First, we get the object's address, and then get its parent pointer. If the parent pointer is null, it means the object is already detached, so we return.

134a ⟨*Remove object* 134a⟩≡ (207) 134b ▷

```
remove_obj:
    SUBROUTINE

    LDA     OPERAND0                ; obj_ptr = get_object_addr<obj_num>
    JSR     get_object_addr
    LDY     #OBJECT_PARENT_OFFSET   ; A = obj_ptr->parent
    LDA     (SCRATCH2),Y
    BNE     .continue               ; if (!A) return
    RTS

.continue:
```

Defines:
`remove_obj`, used in chunks 197 and 199a.

Uses `OBJECT_PARENT_OFFSET` 206a, `OPERAND0` 204, `SCRATCH2` 204, and `get_object_addr` 132.

Next, we save the object's address on the stack.

134b ⟨*Remove object* 134a⟩+≡ (207) ◁ 134a 134c ▷

```
    TAX                             ; save obj_ptr
    LDA     SCRATCH2
    PHA
    LDA     SCRATCH2+1
    PHA
    TXA
```

Uses `SCRATCH2` 204.

Next, we get the parent's first child pointer.

134c ⟨*Remove object* 134a⟩+≡ (207) ◁ 134b 135a ▷

```
    JSR     get_object_addr         ; parent_ptr = get_object_addr<A>
    LDY     #OBJECT_CHILD_OFFSET    ; child_num = parent_ptr->child
    LDA     (SCRATCH2),Y
```

Uses `OBJECT_CHILD_OFFSET` 206a, `SCRATCH2` 204, and `get_object_addr` 132.

If the first child pointer isn't the object we want to detach, then we will need to traverse the children list to find it.

135a ⟨*Remove object* 134a⟩+≡ (207) ◁134c 135b▷

```
    CMP     OPERAND0               ; if (child_num != obj_num) loop
    BNE     .loop
```

Uses OPERAND0 204.

But otherwise, we get the object's sibling and replace the parent's first child with it.

135b ⟨*Remove object* 134a⟩+≡ (207) ◁135a 136a▷

```
    PLA                            ; restore obj_ptr
    STA     SCRATCH1+1
    PLA
    STA     SCRATCH1
    LDA     SCRATCH1
    PHA
    LDA     SCRATCH1+1
    PHA
    LDY     #OBJECT_SIBLING_OFFSET ; A = obj_ptr->next
    LDA     (SCRATCH1),Y
    LDY     #OBJECT_CHILD_OFFSET   ; parent_ptr->child = A
    STA     (SCRATCH2),Y
    JMP     .detach
```

Uses OBJECT_CHILD_OFFSET 206a, OBJECT_SIBLING_OFFSET 206a, SCRATCH1 204, and SCRATCH2 204.

Detaching the object means we null out the parent pointer of the object. Then we can return.

135c ⟨*Detach object* 135c⟩≡ (136b)

```
  .detach:
    PLA                            ; restore obj_ptr
    STA     SCRATCH2+1
    PLA
    STA     SCRATCH2
    LDY     #OBJECT_PARENT_OFFSET  ; obj_ptr->parent = 0
    LDA     #$00
    STA     (SCRATCH2),Y
    INY
    STA     (SCRATCH2),Y
    RTS
```

Uses OBJECT_PARENT_OFFSET 206a and SCRATCH2 204.

Looping over the children just involves traversing the children list and checking if the current child pointer is equal to the object we want to detach. For a self-consistent table, an object's parent must contain the object as a child, and so it would have to be found at some point.

136a ⟨*Remove object* 134a⟩+≡ (207) ◁135b 136b▷

```
.loop:
    JSR     get_object_addr        ; child_ptr = get_object_addr<child_num>
    LDY     #OBJECT_SIBLING_OFFSET ; child_num = child_ptr->next
    LDA     (SCRATCH2),Y
    CMP     OPERAND0               ; if (child_num != obj_num) loop
    BNE     .loop
```

Uses OBJECT_SIBLING_OFFSET 206a, OPERAND0 204, SCRATCH2 204, and get_object_addr 132.

SCRATCH2 now contains the address of the child whose sibling is the object we want to detach. So, we set SCRATCH1 to the object we want to detach, get its sibling, and set it as the sibling of the SCRATCH2 object. Then we can detach the object.

Diagram this.

136b ⟨*Remove object* 134a⟩+≡ (207) ◁136a

```
    PLA                            ; restore obj_ptr
    STA     SCRATCH1+1
    PLA
    STA     SCRATCH1
    LDA     SCRATCH1
    PHA
    LDA     SCRATCH1+1
    PHA
    LDA     (SCRATCH1),Y           ; child_ptr->next = obj_ptr->next
    STA     (SCRATCH2),Y

  ⟨Detach object 135c⟩
```

Uses SCRATCH1 204 and SCRATCH2 204.

12.4 Object strings

The `print_obj_in_A` routine prints the short name of the object in the `A` register. The short name of an object is stored at the beginning of the object's properties as a length-prefixed z-encoded string. The length is actually the number of words, not bytes or characters, and is a single byte. This means that the number of bytes in the string is at most `255*2=510`. And since z-encoded characters are encoded as three characters for every two bytes, the number of characters in a short name is at most `255*3=765`.

137 ⟨*Print object in A* 137⟩≡ (207)

```
print_obj_in_A:
    JSR     get_object_addr         ; obj_ptr = get_object_addr<A>
    LDY     #OBJECT_PROPS_OFFSET    ; props_ptr = obj_ptr->props
    LDA     (SCRATCH2),Y
    STA     SCRATCH1+1
    INY
    LDA     (SCRATCH2),Y
    STA     SCRATCH1
    MOVW    SCRATCH1, SCRATCH2
    INCW    SCRATCH2                ; ++props_ptr
    JSR     load_address            ; Z_PC2 = props_ptr
    JMP     print_zstring           ; print_zstring<Z_PC2>
```

Defines:
  `print_obj_in_A`, used in chunks 69 and 186b.
Uses `INCW` 13b, `MOVW` 12a, `OBJECT_PROPS_OFFSET` 206a, `SCRATCH1` 204, `SCRATCH2` 204, `get_object_addr` 132, `load_address` 45b, and `print_zstring` 62.

12.5 Object attributes

The attributes of an object are stored in the first 4 bytes of the object in the object table. These were also called "flags" in the original Infocom source code, and as such, attributes are binary flags. The order of attributes in these bytes is such that attribute 0 is in bit 7 of byte 0, and attribute 31 is in bit 0 of byte 3.

The `attr_ptr_and_mask` routine is used in attribute instructions to get the pointer to the attributes for the object in `OPERAND0` and mask for the attribute number in `OPERAND1`.

The result from this routine is that `SCRATCH1` contains the relevant attribute word, `SCRATCH3` contains the relevant attribute mask, and `SCRATCH2` contains the address of the attribute word.

We first set `SCRATCH2` to point to the 2-byte word containing the attribute.

138 ⟨*Get attribute pointer and mask* 138⟩≡ (207) 139a ▷

```
attr_ptr_and_mask:
    LDA     OPERAND0            ; SCRATCH2 = get_object_addr<obj_num>
    JSR     get_object_addr
    LDA     OPERAND1            ; if (attr_num >= #$10) {
    CMP     #$10                ;  SCRATCH2 += 2; attr_num -= #$10
    BCC     .continue2          ; }
    SEC
    SBC     #$10
    INCW    SCRATCH2
    INCW    SCRATCH2

.continue2:
    STA     SCRATCH1            ; SCRATCH1 = attr_num
```

Defines:
`attr_ptr_and_mask`, used in chunks 181b, 187, and 199b.

Uses `INCW` 13b, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, and `get_object_addr` 132.

Next, we set `SCRATCH3` to `#$0001` and then bit-shift left by 15 minus the attribute (mod 16) that we want. Thus, attribute 0 and attribute 16 will result in `#$8000`.

139a ⟨*Get attribute pointer and mask* 138⟩+≡ (207) ◁138 139b▷

```
    STOW     #$0001, SCRATCH3
    LDA      #$0F
    SEC
    SBC      SCRATCH1
    TAX

.shift_loop:
    BEQ      .done_shift
    ASL      SCRATCH3
    ROL      SCRATCH3+1
    DEX
    JMP      .shift_loop

.done_shift:
```

Uses `SCRATCH1` 204, `SCRATCH3` 204, and `STOW` 10.

Finally, we load the attribute word into `SCRATCH1`.

139b ⟨*Get attribute pointer and mask* 138⟩+≡ (207) ◁139a

```
    LDY      #$00
    LDA      (SCRATCH2),Y
    STA      SCRATCH1+1
    INY
    LDA      (SCRATCH2),Y
    STA      SCRATCH1
    RTS
```

Uses `SCRATCH1` 204 and `SCRATCH2` 204.

12.6 Object properties

The pointer to the properties of an object is stored in the last 2 bytes of the object in the object table. The first "property" is actually the object's short name, as detailed in Object strings.

Each property starts with a size byte, which is encoded with the lower 5 bits being the property number, and the upper 3 bits being the data size minus 1 (so 0 means 1 byte and 7 means 8 bytes). The property numbers are ordered from lowest to highest for more efficient searching.

The `get_property_ptr` routine gets the pointer to the property table for the object in `OPERAND0` and stores it in `SCRATCH2`. In addition, it returns the size of the first "property" (the short name) in the `Y` register, so that `SCRATCH2+Y` would point to the first numbered property.

140 ⟨*Get property pointer* 140⟩≡ (207)

```
get_property_ptr:
    SUBROUTINE

    LDA     OPERAND0
    JSR     get_object_addr
    LDY     #OBJECT_PROPS_OFFSET
    LDA     (SCRATCH2),Y
    STA     SCRATCH1+1
    INY
    LDA     (SCRATCH2),Y
    STA     SCRATCH1
    ADDW    SCRATCH1, Z_HEADER_ADDR, SCRATCH2
    LDY     #$00
    LDA     (SCRATCH2),Y
    ASL
    TAY
    INY
    RTS
```

Defines:
`get_property_ptr`, used in chunks 189, 191, 194, and 198.

Uses `ADDW` 15c, `OBJECT_PROPS_OFFSET` 206a, `OPERAND0` 204, `SCRATCH1` 204, `SCRATCH2` 204, and `get_object_addr` 132.

The `get_property_num` routine gets the property number being currently pointed to.

141a ⟨*Get property number* 141a⟩≡ (207)

```
get_property_num:
    SUBROUTINE

    LDA      (SCRATCH2),Y
    AND      #$1F
    RTS
```

Defines:
`get_property_num`, used in chunks 189, 191, 194, and 198.
Uses `SCRATCH2` 204.

The `get_property_len` routine gets the length of the property being currently pointed to, minus one.

141b ⟨*Get property length* 141b⟩≡ (207)

```
get_property_len:
    SUBROUTINE

    LDA      (SCRATCH2),Y
    ROR
    ROR
    ROR
    ROR
    ROR
    AND      #$07
    RTS
```

Defines:
`get_property_len`, used in chunks 142, 193, 195, and 198.
Uses `SCRATCH2` 204.

The `next_property` routine updates the `Y` register to point to the next property in the property table.

142 ⟨*Next property* 142⟩≡ (207)

```
next_property:
    SUBROUTINE

    JSR      get_property_len
    TAX

.loop:
    INY
    DEX
    BPL      .loop
    INY
    RTS
```

Defines:
`next_property`, used in chunks 189, 191, 194, and 198.

Uses `get_property_len` 141b.

Chapter 13

Saving and restoring the game

13.0.1 Save prompts for the user

The first part of saving the game asks the user to insert a save diskette, along with the save number (0-7), the drive slot (1-7), and the drive number (1 or 2) containing the save disk.

We first prompt the user to insert the disk:

143 ⟨*Insert save diskette* 143⟩≡ (207) 144a ▷

```
please_insert_save_diskette:
    SUBROUTINE

    JSR     home
    JSR     dump_buffer_with_more
    JSR     dump_buffer_with_more
    STOW    sPleaseInsert, SCRATCH2
    LDX     #28
    JSR     print_ascii_string
    JSR     dump_buffer_with_more
```

Defines:
`please_insert_save_diskette`, used in chunks 149a and 152b.

Uses `SCRATCH2` 204, `STOW` 10, `dump_buffer_with_more` 54, `home` 48, `print_ascii_string` 59b, and `sPleaseInsert` 144b.

Next, we prompt the user for what position they want to save into. The number must be between 0 and 7, otherwise the user is asked again.

144a ⟨*Insert save diskette* 143⟩+≡ (207) ◁143 146a▷

```
.get_position_from_user:
    LDA     #(sPositionPrompt-sSlotPrompt)
    STA     prompt_offset
    JSR     get_prompted_number_from_user
    CMP     #'0
    BCC     .get_position_from_user
    CMP     #'8
    BCS     .get_position_from_user
    STA     save_position
    JSR     buffer_char
```

Uses `buffer_char` 57, `prompt_offset` 144b, `sPositionPrompt` 144b, `sSlotPrompt` 144b, and `save_position` 144b.

144b ⟨*Save diskette strings* 144b⟩≡ (207)

```
sPleaseInsert:
    DC      "PLEASE INSERT SAVE DISKETTE,"
prompt_offset:
    DC      0
sSlotPrompt:
    DC      "SLOT     (1-7):"
save_slot:
    DC      '6
sDrivePrompt:
    DC      "DRIVE    (1-2):"
save_drive:
    DC      '2
sPositionPrompt:
    DC      "POSITION (0-7):"
save_position:
    DC      '0
sDefault:
    DC      "DEFAULT = "
sReturnToBegin:
    DC      "--- PRESS 'RETURN' KEY TO BEGIN ---"
```

Defines:
- `prompt_offset`, used in chunks 144–46.
- `sDrivePrompt`, used in chunk 146b.
- `sPleaseInsert`, used in chunk 143.
- `sPositionPrompt`, used in chunk 144a.
- `sReturnToBegin`, used in chunk 147a.
- `sSlotPrompt`, used in chunks 144–46.
- `save_drive`, used in chunk 146b.
- `save_position`, used in chunks 144a and 147b.
- `save_slot`, used in chunks 145 and 146a.

The `get_prompted_number_from_user` routine takes an offset from the sSlotPrompt symbol in `prompt_offset`. This offset must point to a 15-character prompt. The routine will print the prompt along with its default value (the byte after the prompt), get a single digit from the user, and then store that back into the default value.

145 ⟨*Get prompted number from user* 145⟩≡ (207)

```
get_prompted_number_from_user:
    SUBROUTINE

    JSR     dump_buffer_with_more
    STOW    sSlotPrompt, SCRATCH2       ; print prompt
    ADDB    SCRATCH2, prompt_offset
    LDX     #15
    JSR     print_ascii_string
    JSR     dump_buffer_line
    LDA     #25
    STA     CH
    LDA     #$3F                        ; set inverse
    STA     INVFLG
    STOW    sDefault, SCRATCH2          ; print "DEFAULT = "
    LDX     #10
    JSR     cout_string
    STOW    save_slot, SCRATCH2         ; print default value
    ADDB    SCRATCH2, prompt_offset
    LDX     #1
    JSR     cout_string
    LDA     #$FF                        ; clear inverse
    STA     INVFLG
    JSR     RDKEY                       ; A = read key
    PHA
    LDA     #25
    STA     CH
    JSR     CLREOL                      ; clear line
    PLA
    CMP     #$8D                        ; newline?
    BNE     .end
    LDY     prompt_offset               ; store result
    LDA     save_slot,Y

.end:
    AND     #$7F
    RTS
```

Uses `ADDB` 15a, `CH` 203, `CLREOL` 203, `INVFLG` 203, `RDKEY` 203, `SCRATCH2` 204, `STOW` 10, `cout_string` 47, `dump_buffer_line` 53, `dump_buffer_with_more` 54, `print_ascii_string` 59b, `prompt_offset` 144b, `sSlotPrompt` 144b, and `save_slot` 144b.

Getting back to the save procedure, we then ask the user for the drive slot, which must be between 1 and 7. We also store the slot times 16 in iob.slot_times_16.

146a ⟨*Insert save diskette* 143⟩+≡ (207) ◁144a 146b▷

```
.get_slot_from_user:
    LDA      #(sSlotPrompt - sSlotPrompt)
    STA      prompt_offset
    JSR      get_prompted_number_from_user
    CMP      #'1
    BCC      .get_slot_from_user
    CMP      #'8
    BCS      .get_slot_from_user
    TAX
    AND      #$07
    ASL
    ASL
    ASL
    ASL
    STA      iob.slot_times_16
    TXA
    STA      save_slot
    JSR      buffer_char
```

Uses buffer_char 57, iob 105, prompt_offset 144b, sSlotPrompt 144b, and save_slot 144b.

Next, we ask the user for the drive number, which must be 1 or 2. This value is stored in iob.drive.

146b ⟨*Insert save diskette* 143⟩+≡ (207) ◁146a 147a▷

```
.get_drive_from_user:
    LDA      #(sDrivePrompt - sSlotPrompt)
    STA      prompt_offset
    JSR      get_prompted_number_from_user
    CMP      #'1
    BCC      .get_drive_from_user
    CMP      #'3
    BCS      .get_drive_from_user
    TAX
    AND      #$03
    STA      iob.drive
    TXA
    STA      save_drive
    JSR      buffer_char
```

Uses buffer_char 57, iob 105, prompt_offset 144b, sDrivePrompt 144b, sSlotPrompt 144b, and save_drive 144b.

Next, we prompt the user to start.

147a ⟨*Insert save diskette* 143⟩+≡ (207) ◁146b 147b▷

```
.press_return_key_to_begin:
    JSR       dump_buffer_with_more
    STOW      sReturnToBegin, SCRATCH2
    LDX       #35
    JSR       print_ascii_string
    JSR       dump_buffer_line
    JSR       RDKEY
    CMP       #$8D
    BNE       .press_return_key_to_begin
```

Uses RDKEY 203, SCRATCH2 204, STOW 10, dump_buffer_line 53, dump_buffer_with_more 54, print_ascii_string 59b, and sReturnToBegin 144b.

SCRATCH1 is going to contain 64 * save_position - 1 at the end of the routine. This is the sector number (minus one) where the save data will be written. Thus, a save game takes 64 sectors.

147b ⟨*Insert save diskette* 143⟩+≡ (207) ◁147a

```
    LDA       #$FF
    STA       SCRATCH1
    STA       SCRATCH1+1
    LDA       save_position
    AND       #$07
    BEQ       .end
    TAY

.loop:
    ADDB      SCRATCH1, #64
    DEY
    BNE       .loop

.end:
    JSR       dump_buffer_with_more
    RTS
```

Uses ADDB 15a, SCRATCH1 204, dump_buffer_with_more 54, and save_position 144b.

When the save is eventually complete, the user is prompted to reinsert the game diskette.

148 ⟨*Reinsert game diskette* 148⟩≡ (207)

```
sReinsertGameDiskette:
    DC      "PLEASE RE-INSERT GAME DISKETTE,"
sPressReturnToContinue:
    DC      "--- PRESS 'RETURN' KEY TO CONTINUE ---"

please_reinsert_game_diskette:
    SUBROUTINE

    LDA     iob.slot_times_16
    CMP     #$60
    BNE     .set_slot6_drive1
    LDA     iob.drive
    CMP     #$01
    BNE     .set_slot6_drive1
    JSR     dump_buffer_with_more
    STOW    sReinsertGameDiskette, SCRATCH2
    LDX     #31
    JSR     print_ascii_string

.await_return_key:
    JSR     dump_buffer_with_more
    STOW    sPressReturnToContinue, SCRATCH2
    LDX     #38
    JSR     print_ascii_string
    JSR     dump_buffer_line
    JSR     RDKEY
    CMP     #$8D
    BNE     .await_return_key
    JSR     dump_buffer_with_more

.set_slot6_drive1:
    LDA     #$60
    STA     iob.slot_times_16
    LDA     #$01
    STA     iob.drive
    RTS
```

Defines:
- `please_reinsert_game_diskette`, used in chunks 151c, 152a, and 155.
- `sPressReturnToContinue`, never used.
- `sReinsertGameDiskette`, never used.

Uses `RDKEY` 203, `SCRATCH2` 204, `STOW` 10, `dump_buffer_line` 53, `dump_buffer_with_more` 54, `iob` 105, and `print_ascii_string` 59b.

13.0.2 Saving the game state

When the virtual machine is instructed to save, the `instr_save` routine is execute.

The instruction first calls the `please_insert_save_diskette` routine to prompt the user to insert a save diskette and set the disk parameters.

149a ⟨*Instruction save* 149a⟩≡ (207) 149b ▷

```
instr_save:
    SUBROUTINE

    JSR     please_insert_save_diskette
```

Defines:
`instr_save`, used in chunk 109.
Uses `please_insert_save_diskette` 143.

Next, we store the z-machine version number to the first byte of the `BUFF_AREA`. We maintain a pointer into the buffer in the `X` register.

149b ⟨*Instruction save* 149a⟩+≡ (207) ◁ 149a 149c ▷

```
    LDX     #$00
    LDY     #$00
    LDA     (Z_HEADER_ADDR),Y
    STA     BUFF_AREA,X
    INX
```

Uses `BUFF_AREA` 204.

Next, we copy the 3 bytes of `Z_PC` to the buffer. This is actually done in reverse order.

149c ⟨*Instruction save* 149a⟩+≡ (207) ◁ 149b 150b ▷

```
    STOW    #Z_PC, SCRATCH2
    LDY     #$03
    JSR     copy_data_to_buff
```

Uses `SCRATCH2` 204, `STOW` 10, `Z_PC` 204, and `copy_data_to_buff` 150a.

The `copy_data_to_buff` routine copies the number of bytes in the `Y` register from the address in `SCRATCH2` to the buffer, updating `X` as the pointer into the buffer.

150a ⟨*Copy data to buff* 150a⟩≡ (207)

```
copy_data_to_buff:
    SUBROUTINE

    DEY
    LDA      (SCRATCH2),Y
    STA      BUFF_AREA,X
    INX
    CPY      #$00
    BNE      copy_data_to_buff
    RTS
```

Defines:
`copy_data_to_buff`, used in chunks 149–51.
Uses `BUFF_AREA` 204 and `SCRATCH2` 204.

We copy the 30 bytes of the `LOCAL_ZVARS` to the buffer, then 6 bytes for the stack state starting from `STACK_COUNT`. The collected buffer is then written to the first save sector on disk.

150b ⟨*Instruction save* 149a⟩+≡ (207) ◁149c 151a▷

```
    STOW     #LOCAL_ZVARS, SCRATCH2
    LDY      #30
    JSR      copy_data_to_buff

    STOW     #STACK_COUNT, SCRATCH2
    LDY      #6
    JSR      copy_data_to_buff

    JSR      write_next_sector
    BCS      .fail
```

Uses `LOCAL_ZVARS` 204, `SCRATCH2` 204, `STACK_COUNT` 204, `STOW` 10, `copy_data_to_buff` 150a, and `write_next_sector` 108.

The second sector written contains 256 bytes starting from #$0280, and the third sector contains 256 bytes starting from #$0380.

151a ⟨*Instruction save* 149a⟩+≡ (207) ◁150b 151b▷
```
    LDX     #$00
    STOW    #$0280, SCRATCH2
    LDY     #$00
    JSR     copy_data_to_buff

    JSR     write_next_sector
    BCS     .fail

    LDX     #$00
    STOW    #$0380, SCRATCH2
    LDY     #$68
    JSR     copy_data_to_buff

    JSR     write_next_sector
    BCS     .fail
```
Uses SCRATCH2 204, STOW 10, copy_data_to_buff 150a, and write_next_sector 108.

Next, we write the game memory starting from Z_HEADER_ADDR all the way up to the base of static memory given by the header.

151b ⟨*Instruction save* 149a⟩+≡ (207) ◁151a 151c▷
```
    MOVW    Z_HEADER_ADDR, SCRATCH2
    LDY     #HEADER_STATIC_MEM_BASE
    LDA     (Z_HEADER_ADDR),Y
    STA     SCRATCH3                ; big-endian!
    INC     SCRATCH3

.loop:
    JSR     inc_sector_and_write
    BCS     .fail
    INC     SCRATCH2+1
    DEC     SCRATCH3
    BNE     .loop
    JSR     inc_sector_and_write
    BCS     .fail
```
Uses HEADER_STATIC_MEM_BASE 206a, MOVW 12a, SCRATCH2 204, SCRATCH3 204, and inc_sector_and_write 108.

Finally, we ask the user to reinsert the game diskette, and we're done. The instruction branches, assuming success.

151c ⟨*Instruction save* 149a⟩+≡ (207) ◁151b 152a▷
```
    JSR     please_reinsert_game_diskette
    JMP     branch
```
Uses branch 161a and please_reinsert_game_diskette 148.

On failure, the instruction also asks the user to reinsert the game diskette, but branches assuming failure.

152a ⟨*Instruction save* 149a⟩+≡ (207) ◁151c

```
  .fail:
      JSR      please_reinsert_game_diskette
      JMP      negated_branch
```

Uses `negated_branch` 161a and `please_reinsert_game_diskette` 148.

13.0.3 Restoring the game state

When the virtual machine is instructed to restore, the `instr_restore` routine is executed. The instruction starts by asking the user to insert the save diskette, and sets up the disk parameters.

152b ⟨*Instruction restore* 152b⟩≡ (207) 152c▷

```
  instr_restore:
      SUBROUTINE

      JSR      please_insert_save_diskette
```

Defines:
`instr_restore`, used in chunk 109.
Uses `please_insert_save_diskette` 143.

The next step is to read the first sector and check the z-machine version number to make sure it's the same as the currently executing z-machine version. Otherwise the instruction fails.

152c ⟨*Instruction restore* 152b⟩+≡ (207) ◁152b 153a▷

```
      JSR      read_next_sector
      BCC      .continue
      JMP      .fail

  .continue:
      LDX      #$00
      LDY      #$00
      LDA      (Z_HEADER_ADDR),Y
      CMP      BUFF_AREA,X
      BEQ      .continue2
      JMP      .fail
```

Uses `BUFF_AREA` 204 and `read_next_sector` 107.

We also save the current game flags in the header at byte `#$11`.

153a ⟨*Instruction restore* 152b⟩+≡ (207) ◁152c 153b▷

```
.continue2:
    LDY      #$11                   ; Game flags.
    LDA      (Z_HEADER_ADDR),Y
    STA      SIGN_BIT
```

We then restore the `Z_PC`, local variables, and stack state from the same sector.

153b ⟨*Instruction restore* 152b⟩+≡ (207) ◁153a 154a▷

```
    INX
    STOW     #Z_PC, SCRATCH2
    LDY      #3
    JSR      copy_data_from_buff
    LDA      #$00
    STA      ZCODE_PAGE_VALID
    STOW     #LOCAL_ZVARS, SCRATCH2
    LDY      #30
    JSR      copy_data_from_buff
    STOW     #STACK_COUNT, SCRATCH2
    LDY      #6
    JSR      copy_data_from_buff
```

Uses `LOCAL_ZVARS` 204, `SCRATCH2` 204, `STACK_COUNT` 204, `STOW` 10, `ZCODE_PAGE_VALID` 204, `Z_PC` 204, and `copy_data_from_buff` 153c.

The `copy_data_from_buff` routine copies the number of bytes in the `Y` register from `BUFF_AREA` to the address in `SCRATCH2`, updating `X` as the pointer into the buffer.

153c ⟨*Copy data from buff* 153c⟩≡ (207)

```
copy_data_from_buff:
    SUBROUTINE

    DEY
    LDA      BUFF_AREA,X
    STA      (SCRATCH2),Y
    INX
    CPY      #$00
    BNE      copy_data_from_buff
    RTS
```

Defines:
`copy_data_from_buff`, used in chunks 153b and 154a.
Uses `BUFF_AREA` 204 and `SCRATCH2` 204.

Next we restore 256 bytes starting from #$0280 from the second sector, and 256 bytes starting from #$0380 from the third sector.

154a ⟨*Instruction restore* 152b⟩+≡ (207) ◁153b 154b▷

```
    JSR     read_next_sector
    BCS     .fail
    LDX     #$00
    STOW    #$0280, SCRATCH2
    LDY     #$00
    JSR     copy_data_from_buff
    JSR     read_next_sector
    BCS     .fail
    LDX     #$00
    STOW    #$0380, SCRATCH2
    LDY     #$68
    JSR     copy_data_from_buff
```

Uses SCRATCH2 204, STOW 10, copy_data_from_buff 153c, and read_next_sector 107.

Next, we restore the game memory starting from Z_HEADER_ADDR all the way up to the base of static memory given by the header.

154b ⟨*Instruction restore* 152b⟩+≡ (207) ◁154a 154c▷

```
    MOVW    Z_HEADER_ADDR, SCRATCH2
    LDY     #$0E
    LDA     (Z_HEADER_ADDR),Y
    STA     SCRATCH3            ; big-endian!
    INC     SCRATCH3

.loop:
    JSR     inc_sector_and_read
    BCS     .fail
    INC     SCRATCH2+1
    DEC     SCRATCH3
    BNE     .loop
```

Uses MOVW 12a, SCRATCH2 204, SCRATCH3 204, and inc_sector_and_read 107.

Then we restore the game flags in the header at byte #$11 from before the actual restore.

154c ⟨*Instruction restore* 152b⟩+≡ (207) ◁154b 155a▷

```
    LDA     SIGN_BIT
    LDY     #$11
    STA     (Z_HEADER_ADDR),Y
```

Finally, we ask the user to reinsert the game diskette, and we're done. The instruction branches, assuming success.

155a ⟨*Instruction restore* 152b⟩+≡ (207) ◁154c 155b▷

```
      JSR      please_reinsert_game_diskette
      JMP      branch
```

Uses `branch` 161a and `please_reinsert_game_diskette` 148.

On failure, the instruction also asks the user to reinsert the game diskette, but branches assuming failure.

155b ⟨*Instruction restore* 152b⟩+≡ (207) ◁155a

```
  .fail:
      JSR      please_reinsert_game_diskette
      JMP      negated_branch
```

Uses `negated_branch` 161a and `please_reinsert_game_diskette` 148.

Chapter 14

Instructions

After an instruction finishes, it must jump to `do_instruction` in order to execute the next instruction.

Note that return values from functions are always stored in `OPERAND0`.

Data movement instructions	
`load`	Loads a variable into a variable
`loadb`	Loads a byte from a byte array into a variable
`loadb`	Loads a word from a word array into a variable
`store`	Stores a value into a variable
`storeb`	Stores a byte into a byte array
`storew`	Stores a word into a word array
Stack instructions	
`pop`	Throws away the top item from the stack
`pull`	Pulls a value from the stack into a variable
`push`	Pushes a value onto the stack
Decrement/increment instructions	
`dec`	Decrements a variable
`inc`	Increments a variable
Arithmetic instructions	
`add`	Adds two signed 16-bit values, storing to a variable
`div`	Divides two signed 16-bit values, storing to a variable
`mod`	Modulus of two signed 16-bit values, storing to a variable
`mul`	Multiplies two signed 16-bit values, storing to a variable
`random`	Stores a random number to a variable

`sub`	Subtracts two signed 16-bit values, storing to a variable
	Logical instructions
`and`	Bitwise ANDs two 16-bit values, storing to a variable
`not`	Bitwise NOTs two 16-bit values, storing to a variable
`or`	Bitwise ORs two 16-bit values, storing to a variable
	Conditional branch instructions
`dec_chk`	Decrements a variable then branches if less than value
`inc_chk`	Increments a variable then branches if greater than value
`je`	Branches if value is equal to any subsequent operand
`jg`	Branches if value is (signed) greater than second operand
`jin`	Branches if object is a direct child of second operand object
`jl`	Branches if value is (signed) less than second operand
`jz`	Branches if value is equal to zero
`test`	Branches if all set bits in first operand are set in second operand
`test_attr`	Branches if object has attribute in second operand set
	Jump and subroutine instructions
`call`	Calls a subroutine
`jump`	Jumps unconditionally
`print_ret`	Prints a string and returns true
`ret`	Returns a value
`ret_popped`	Returns the popped value from the stack
`rfalse`	Returns false
`rtrue`	Returns true
	Print instructions
`new_line`	Prints a newline
`print`	Prints the immediate string
`print_addr`	Prints the string at an address
`print_char`	Prints the immediate character
`print_num`	Prints the signed number
`print_obj`	Prints the object's short name
`print_paddr`	Prints the string at a packed address
	Object instructions
`clear_attr`	Clears an object's attribute
`get_child`	Stores the object's first child into a variable
`get_next_prop`	Stores the object's property number after the given property number into a variable
`get_parent`	Stores the object's parent into a variable
`get_prop`	Stores the value of the object's property into a variable
`get_prop_addr`	Stores the address of the object's property into a variable
`get_prop_len`	Stores the byte length of the object's property into a variable
`get_sibling`	Stores the next sibling of the object into a variable
`insert_obj`	Reparents the object to the destination object
`put_prop`	Stores the value into the object's property

`remove_obj`	Detaches the object from its parent
`set_attr`	Sets an object's attribute
	Other instructions
`nop`	Does nothing
`restart`	Restarts the game
`restore`	Loads a saved game
`quit`	Quits the game
`save`	Saves the game
`sread`	Reads from the keyboard

14.1 Instruction utilities

There are a few utilities that are used in common by instructions.

`illegal_opcode` hits a `BRK` instruction.

158 ⟨*Instruction illegal opcode* 158⟩≡ (207)

```
illegal_opcode:
    SUBROUTINE

    JSR     brk
```

Defines:
`illegal_opcode`, used in chunks 109, 113b, 115a, 117b, and 119.
Uses `brk` 31c.

The `store_zero_and_next` routine stores the value `0` into the variable in the next byte, while `store_A_and_next` stores the value in the `A` register into the variable in in the next byte. Finally, `store_and_next` stores the value in `SCRATCH2` into the variable in the next byte.

159a ⟨*Store and go to next instruction* 159a⟩≡ (207)

```
store_zero_and_next:
    SUBROUTINE

    LDA      #$00

store_A_and_next:
    SUBROUTINE

    STA      SCRATCH2
    LDA      #$00
    STA      SCRATCH2+1

store_and_next:
    SUBROUTINE

    JSR      store_var
    JMP      do_instruction
```

Defines:
`store_A_and_next`, used in chunks 189 and 195.
`store_and_next`, used in chunks 125, 131, 165, 166a, 170b, 172–76, 190, and 192–94.
`store_zero_and_next`, used in chunks 189 and 194.
Uses `SCRATCH2` 204, `do_instruction` 112, and `store_var` 123.

The `print_zstring_and_next` routine prints the z-encoded string at `Z_PC2` to the screen, and then goes to the next instruction.

159b ⟨*Print zstring and go to next instruction* 159b⟩≡ (207)

```
print_zstring_and_next:
    SUBROUTINE

    JSR      print_zstring
    JMP      do_instruction
```

Defines:
`print_zstring_and_next`, used in chunks 185b and 186c.
Uses `do_instruction` 112 and `print_zstring` 62.

The `inc_var` routine increments the variable in `OPERAND0`, and also stores the result in `SCRATCH2`.

160a ⟨*Increment variable* 160a⟩≡ (207)

```
inc_var:
    SUBROUTINE

    LDA     OPERAND0
    JSR     var_get
    INCW    SCRATCH2
inc_var_continue:
    PSHW    SCRATCH2
    LDA     OPERAND0
    JSR     var_put
    PULW    SCRATCH2
    RTS
```

Defines:
`inc_var`, used in chunks 169c and 177a.
Uses `INCW` 13b, `OPERAND0` 204, `PSHW` 12b, `PULW` 13a, `SCRATCH2` 204, `var_get` 122, and `var_put` 124.

`dec_var` does the same thing as `inc_var`, except does a decrement.

160b ⟨*Decrement variable* 160b⟩≡ (207)

```
dec_var:
    SUBROUTINE

    LDA     OPERAND0
    JSR     var_get
    SUBB    SCRATCH2, #$01
    JMP     inc_var_continue
```

Defines:
`dec_var`, used in chunks 170a and 176b.
Uses `OPERAND0` 204, `SCRATCH2` 204, `SUBB` 16b, and `var_get` 122.

14.1.1 Handling branches

Branch information is stored in one or two bytes, indicating what to do with the result of the test. If bit `7` of the first byte is `0`, a branch occurs when the condition was false; if `1`, then branch is on true.

There are two entry points here, `branch` and `negated_branch`, which are used when the branch condition previously checked is true and false, respectively.

`branch` checks if bit `7` of the offset data is clear, and if so, does the branch, otherwise skips to the next instruction.

`negated_branch` is the same, except that it inverts the branch condition.

161a ⟨*Handle branch* 161a⟩≡ (207) 161b ▷

```
negated_branch:
    SUBROUTINE

    JSR       get_next_code_byte
    ORA       #$00
    BMI       .do_branch
    BPL       .no_branch

branch:
    JSR       get_next_code_byte
    ORA       #$00
    BPL       .do_branch
```

Defines:
`branch`, used in chunks 151c, 155a, 177, 178, and 180b.
`negated_branch`, used in chunks 152a, 155b, 177–81, and 188.
Uses `get_next_code_byte` 37.

If we're not branching, we check whether bit `6` is set. If so, we need to read the second byte of the offset data and throw it away. In either case, we go to the next instruction.

161b ⟨*Handle branch* 161a⟩+≡ (207) ◁ 161a 162 ▷

```
.no_branch:
    AND       #$40
    BNE       .next
    JSR       get_next_code_byte

.next:
    JMP       do_instruction
```

Uses `do_instruction` 112 and `get_next_code_byte` 37.

With the first byte of the branch offset data in the `A` register, we check whether bit `6` is set. If so, the offset is (unsigned) 6 bits and we can move on, otherwise we need to tack on the next byte for a signed 14-bit offset. When we're done, `SCRATCH2` will contain the signed offset.

162 ⟨*Handle branch* 161a⟩+≡ (207) ◁161b 163a▷

```
.do_branch:
    TAX
    AND     #$40
    BEQ     .get_14_bit_offset

.offset_is_6_bits:
    TXA
    AND     #$3F
    STA     SCRATCH2
    LDA     #$00
    STA     SCRATCH2+1
    JMP     .check_for_return_false

.get_14_bit_offset:
    TXA
    AND     #$3F
    PHA
    JSR     get_next_code_byte
    STA     SCRATCH2
    PLA
    STA     SCRATCH2+1
    AND     #$20
    BEQ     .check_for_return_false
    LDA     SCRATCH2+1
    ORA     #$C0
    STA     SCRATCH2+1
```

Uses `SCRATCH2` 204 and `get_next_code_byte` 37.

An offset of 0 always means to return false from the current routine, while an offset of 1 means to return true. Otherwise, we fall through.

163a ⟨*Handle branch* 161a⟩+≡ (207) ◁162 163b▷

```
  .check_for_return_false:
      LDA       SCRATCH2+1
      ORA       SCRATCH2
      BEQ       instr_rfalse
      LDA       SCRATCH2
      SEC
      SBC       #$01
      STA       SCRATCH2
      BCS       .check_for_return_true
      DEC       SCRATCH2+1

  .check_for_return_true:
      LDA       SCRATCH2+1
      ORA       SCRATCH2
      BEQ       instr_rtrue
```

Uses `SCRATCH2` 204, `instr_rfalse` 183b, and `instr_rtrue` 184a.

We now need to move execution to the instruction at address `Address after branch data + offset - 2`.

We subtract 1 from the offset in `SCRATCH2`. Note that above, we've already subtracted 1, so now we've subtracted 2 from the offset.

163b ⟨*Handle branch* 161a⟩+≡ (207) ◁163a 163c▷

```
  branch_to_offset:
      SUBROUTINE

      SUBB      SCRATCH2, #$01
```

Defines:
`branch_to_offset`, used in chunk 182a.
Uses `SCRATCH2` 204 and `SUBB` 16b.

Next, we store twice the high byte of `SCRATCH2` into `SCRATCH1`.

163c ⟨*Handle branch* 161a⟩+≡ (207) ◁163b 164▷

```
      LDA       SCRATCH2+1
      STA       SCRATCH1
      ASL
      LDA       #$00
      ROL
      STA       SCRATCH1+1
```

Uses `SCRATCH1` 204 and `SCRATCH2` 204.

Finally, we add the signed 16-bit SCRATCH2 to the 24-bit Z_PC, and go to the next instruction. We invalidate the zcode page if we've passed a page boundary.

Interestingly, although Z_PC is a 24-bit address, we AND the high byte with #$01, meaning that the maximum Z_PC would be #$01FFFF.

164 ⟨*Handle branch* 161a⟩+≡ (207) ◁163c

```
    LDA     Z_PC
    CLC
    ADC     SCRATCH2
    BCC     .continue2
    INC     SCRATCH1
    BNE     .continue2
    INC     SCRATCH1+1

.continue2:
    STA     Z_PC
    LDA     SCRATCH1+1
    ORA     SCRATCH1
    BEQ     .next

    CLC
    LDA     SCRATCH1
    ADC     Z_PC+1
    STA     Z_PC+1
    LDA     SCRATCH1+1
    ADC     Z_PC+2
    AND     #$01
    STA     Z_PC+2
    LDA     #$00
    STA     ZCODE_PAGE_VALID
    JMP     do_instruction

.next:
    JMP     do_instruction
```

Uses SCRATCH1 204, SCRATCH2 204, ZCODE_PAGE_VALID 204, Z_PC 204, and do_instruction 112.

14.2 Data movement instructions

14.2.1 load

`load` loads the variable in the operand into the variable in the next code byte.

165a ⟨*Instruction load* 165a⟩≡ (207)

```
instr_load:
    SUBROUTINE

    LDA     OPERAND0
    JSR     var_get
    JMP     store_and_next
```

Defines:
`instr_load`, used in chunk 109.

Uses `OPERAND0` 204, `store_and_next` 159a, and `var_get` 122.

14.2.2 loadw

`loadw` loads a word from the array at the address given `OPERAND0`, indexed by `OPERAND1`, into the variable in the next code byte.

165b ⟨*Instruction loadw* 165b⟩≡ (207)

```
instr_loadw:
    SUBROUTINE

    ASL     OPERAND1                ; OPERAND1 *= 2
    ROL     OPERAND1+1
    ADDW    OPERAND1, OPERAND0, SCRATCH2
    JSR     load_address
    JSR     get_next_code_word
    JMP     store_and_next
```

Defines:
`instr_loadw`, used in chunk 109.

Uses `ADDW` 15c, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, `get_next_code_word` 45a, `load_address` 45b, and `store_and_next` 159a.

14.2.3 loadb

`loadb` loads a zero-extended byte from the array at the address given `OPERAND0`, indexed by `OPERAND1`, into the variable in the next code byte.

166a ⟨*Instruction loadb* 166a⟩≡ (207)

```
instr_loadb:
    SUBROUTINE

    ADDW     OPERAND1, OPERAND0, SCRATCH2   ; SCRATCH2 = OPERAND0 + OPERAND1
    JSR      load_address                   ; Z_PC2 = SCRATCH2
    JSR      get_next_code_byte2            ; A = *Z_PC2
    STA      SCRATCH2                       ; SCRATCH2 = uint16(A)
    LDA      #$00
    STA      SCRATCH2+1
    JMP      store_and_next                 ; store_and_next(SCRATCH2)
```

Defines:
  `instr_loadb`, used in chunk 109.

Uses `ADDW` 15c, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, `get_next_code_byte2` 43, `load_address` 45b, and `store_and_next` 159a.

14.2.4 store

`store` stores `OPERAND1` into the variable in `OPERAND0`.

166b ⟨*Instruction store* 166b⟩≡ (207)

```
instr_store:
    SUBROUTINE

    MOVW     OPERAND1, SCRATCH2
    LDA      OPERAND0

stretch_var_put:
    JSR      var_put
    JMP      do_instruction
```

Defines:
  `instr_store`, used in chunk 109.
  `stretch_var_put`, used in chunk 169a.

Uses `MOVW` 12a, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, `do_instruction` 112, and `var_put` 124.

14.2.5 storew

`storew` stores `OPERAND2` into the word array pointed to by z-address `OPERAND0` at the index `OPERAND1`.

167 ⟨*Instruction storew* 167⟩≡ (207)

```
instr_storew:
    SUBROUTINE

    LDA     OPERAND1        ; SCRATCH2 = Z_HEADER_ADDR + OPERAND0 + 2*OPERAND1
    ASL
    ROL     OPERAND1+1
    CLC
    ADC     OPERAND0
    STA     SCRATCH2
    LDA     OPERAND1+1
    ADC     OPERAND0+1
    STA     SCRATCH2+1
    ADDW    SCRATCH2, Z_HEADER_ADDR, SCRATCH2
    LDY     #$00
    LDA     OPERAND2+1
    STA     (SCRATCH2),Y
    INY
    LDA     OPERAND2
    STA     (SCRATCH2),Y
    JMP     do_instruction
```

Defines:
`instr_storew`, used in chunk 109.

Uses `ADDW` 15c, `OPERAND0` 204, `OPERAND1` 204, `OPERAND2` 204, `SCRATCH2` 204, and `do_instruction` 112.

14.2.6 storeb

`storeb` stores the low byte of `OPERAND2` into the byte array pointed to by z-address `OPERAND0` at the index `OPERAND1`.

168a ⟨*Instruction storeb* 168a⟩≡ (207)

```
instr_storeb:
    SUBROUTINE

    LDA     OPERAND1        ; SCRATCH2 = Z_HEADER_ADDR + OPERAND0 + OPERAND1
    CLC
    ADC     OPERAND0
    STA     SCRATCH2
    LDA     OPERAND1+1
    ADC     OPERAND0+1
    STA     SCRATCH2+1
    ADDW    SCRATCH2, Z_HEADER_ADDR, SCRATCH2
    LDY     #$00
    LDA     OPERAND2
    STA     (SCRATCH2),Y
    JMP     do_instruction
```

Defines:
`instr_storeb`, used in chunk 109.

Uses `ADDW` 15c, `OPERAND0` 204, `OPERAND1` 204, `OPERAND2` 204, `SCRATCH2` 204, and `do_instruction` 112.

14.3 Stack instructions

14.3.1 pop

`pop` pops the stack. This throws away the popped value.

168b ⟨*Instruction pop* 168b⟩≡ (207)

```
instr_pop:
    SUBROUTINE

    JSR     pop
    JMP     do_instruction
```

Defines:
`instr_pop`, used in chunk 109.

Uses `do_instruction` 112 and `pop` 36.

14.3.2 pull

`pull` pops the top value off the stack and puts it in the variable in `OPERAND0`.

169a ⟨*Instruction pull* 169a⟩≡ (207)

```
instr_pull:
    SUBROUTINE

    JSR     pop
    LDA     OPERAND0
    JMP     stretch_var_put
```

Defines:
`instr_pull`, used in chunk 109.

Uses `OPERAND0` 204, `pop` 36, and `stretch_var_put` 166b.

14.3.3 push

`push` pushes the value in `OPERAND0` onto the z-stack.

169b ⟨*Instruction push* 169b⟩≡ (207)

```
instr_push:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2
    JSR     push
    JMP     do_instruction
```

Defines:
`instr_push`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `SCRATCH2` 204, `do_instruction` 112, and `push` 35.

14.4 Decrements and increments

14.4.1 inc

`inc` increments the variable in the operand.

169c ⟨*Instruction inc* 169c⟩≡ (207)

```
instr_inc:
    SUBROUTINE

    JSR     inc_var
    JMP     do_instruction
```

Defines:
`instr_inc`, used in chunk 109.

Uses `do_instruction` 112 and `inc_var` 160a.

14.4.2 dec

`dec` decrements the variable in the operand.

170a ⟨*Instruction dec* 170a⟩≡ (207)

```
  instr_dec:
      SUBROUTINE

      JSR      dec_var
      JMP      do_instruction
```

Defines:
`instr_dec`, used in chunk 109.

Uses `dec_var` 160b and `do_instruction` 112.

14.5 Arithmetic instructions

14.5.1 add

`add` adds the first operand to the second operand and stores the result in the variable in the next code byte.

170b ⟨*Instruction add* 170b⟩≡ (207)

```
  instr_add:
      SUBROUTINE

      ADDW     OPERAND0, OPERAND1, SCRATCH2
      JMP      store_and_next
```

Defines:
`instr_add`, used in chunk 109.

Uses `ADDW` 15c, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, and `store_and_next` 159a.

14.5.2 div

`div` divides the first operand by the second operand and stores the result in the variable in the next code byte. There are optimizations for dividing by `2` and `4` (which are just shifts). For all other divides, `divu16` is called, and then the sign is adjusted afterwards.

171 ⟨*Instruction div* 171⟩≡ (207)

```
instr_div:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    MOVW     OPERAND1, SCRATCH1
    JSR      check_sign
    LDA      SCRATCH1+1
    BNE      .do_div
    LDA      SCRATCH1
    CMP      #$02
    BEQ      .shortcut_div2
    CMP      #$04
    BEQ      .shortcut_div4

.do_div:
    JSR      divu16
    JMP      stretch_set_sign

.shortcut_div4:
    LSR      SCRATCH2+1
    ROR      SCRATCH2

.shortcut_div2:
    LSR      SCRATCH2+1
    ROR      SCRATCH2
    JMP      stretch_set_sign
```

Defines:
`instr_div`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, `check_sign` 96a, and `divu16` 100.

14.5.3 mod

`mod` divides the first operand by the second operand and stores the remainder in the variable in the next code byte. There are optimizations for dividing by `2` and `4` (which are just shifts). For all other divides, `divu16` is called, and then the sign is adjusted afterwards.

172 ⟨*Instruction mod* 172⟩≡ (207)

```
instr_mod:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    MOVW     OPERAND1, SCRATCH1
    JSR      check_sign
    JSR      divu16
    MOVW     SCRATCH1, SCRATCH2
    JMP      store_and_next
```

Defines:
`instr_mod`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, `check_sign` 96a, `divu16` 100, and `store_and_next` 159a.

14.5.4 mul

`mul` multiplies the first operand by the second operand and stores the result in the variable in the next code byte. There are optimizations for multiplying by `2` and `4` (which are just shifts). For all other multiplies, `mulu16` is called, and then the sign is adjusted afterwards.

173 ⟨*Instruction mul* 173⟩≡ (207)

```
instr_mul:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    MOVW     OPERAND1, SCRATCH1
    JSR      check_sign
    LDA      SCRATCH1+1
    BNE      .do_mult
    LDA      SCRATCH1
    CMP      #$02
    BEQ      .shortcut_x2
    CMP      #$04
    BEQ      .shortcut_x4

.do_mult:
    JSR      mulu16

stretch_set_sign:
    JSR      set_sign
    JMP      store_and_next

.shortcut_x4:
    ASL      SCRATCH2
    ROL      SCRATCH2+1

.shortcut_x2:
    ASL      SCRATCH2
    ROL      SCRATCH2+1
    JMP      stretch_set_sign
```

Defines:
`instr_mul`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, `check_sign` 96a, `mulu16` 97, `set_sign` 96b, and `store_and_next` 159a.

14.5.5 random

`random` gets a random number between 1 and `OPERAND0`.

174a ⟨*Instruction random* 174a⟩≡ (207)

```
instr_random:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH1
    JSR     get_random
    JSR     divu16
    MOVW    SCRATCH1, SCRATCH2
    INCW    SCRATCH2
    JMP     store_and_next
```

Defines:
`instr_random`, used in chunk 109.

Uses `INCW` 13b, `MOVW` 12a, `OPERAND0` 204, `SCRATCH1` 204, `SCRATCH2` 204, `divu16` 100, `get_random` 174b, and `store_and_next` 159a.

174b ⟨*Get random* 174b⟩≡ (207)

```
get_random:
    SUBROUTINE

    ROL     RANDOM_VAL+1
    MOVW    RANDOM_VAL, SCRATCH2
    RTS
```

Defines:
`get_random`, used in chunk 174a.

Uses `MOVW` 12a and `SCRATCH2` 204.

14.5.6 sub

`sub` subtracts the first operand from the second operand and stores the result in the variable in the next code byte.

174c ⟨*Instruction sub* 174c⟩≡ (207)

```
instr_sub:
    SUBROUTINE

    SUBW    OPERAND1, OPERAND0, SCRATCH2
    JMP     store_and_next
```

Defines:
`instr_sub`, used in chunk 109.

Uses `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, `SUBW` 17b, and `store_and_next` 159a.

14.6 Logical instructions

14.6.1 and

`and` bitwise-ands the first operand with the second operand and stores the result in the variable given by the next code byte.

175a ⟨*Instruction and* 175a⟩≡ (207)

```
instr_and:
    SUBROUTINE

    LDA     OPERAND1+1
    AND     OPERAND0+1
    STA     SCRATCH2+1
    LDA     OPERAND1
    AND     OPERAND0
    STA     SCRATCH2
    JMP     store_and_next
```

Defines:
`instr_and`, used in chunk 109.
Uses `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, and `store_and_next` 159a.

14.6.2 not

`not` flips every bit in the variable in the operand and stores it in the variable in the next code byte.

175b ⟨*Instruction not* 175b⟩≡ (207)

```
instr_not:
    SUBROUTINE

    LDA     OPERAND0
    EOR     #$FF
    STA     SCRATCH2
    LDA     OPERAND0+1
    EOR     #$FF
    STA     SCRATCH2+1
    JMP     store_and_next
```

Defines:
`instr_not`, used in chunk 109.
Uses `OPERAND0` 204, `SCRATCH2` 204, and `store_and_next` 159a.

14.6.3 or

`or` bitwise-ors the first operand with the second operand and stores the result in the variable given by the next code byte.

176a ⟨*Instruction or* 176a⟩≡ (207)

```
instr_or:
    SUBROUTINE

    LDA     OPERAND1+1
    ORA     OPERAND0+1
    STA     SCRATCH2+1
    LDA     OPERAND1
    ORA     OPERAND0
    STA     SCRATCH2
    JMP     store_and_next
```

Defines:
  `instr_or`, used in chunk 109.
Uses `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, and `store_and_next` 159a.

14.7 Conditional branch instructions

14.7.1 dec_chk

`dec_chk` decrements the variable in the first operand, and then jumps if it is less than the second operand.

176b ⟨*Instruction dec chk* 176b⟩≡ (207)

```
instr_dec_chk:
    SUBROUTINE

    JSR     dec_var
    MOVW    OPERAND1, SCRATCH1
    JMP     do_chk
```

Defines:
  `instr_dec_chk`, used in chunk 109.
Uses `MOVW` 12a, `OPERAND1` 204, `SCRATCH1` 204, `dec_var` 160b, and `do_chk` 177a.

14.7.2 inc_chk

`inc_chk` increments the variable in the first operand, and then jumps if it is greater than the second operand.

177a ⟨*Instruction inc chk* 177a⟩≡ (207)

```
instr_inc_chk:
    JSR     inc_var
    MOVW    SCRATCH2, SCRATCH1
    MOVW    OPERAND1, SCRATCH2

do_chk:
    JSR     cmp16
    BCC     stretch_to_branch
    JMP     negated_branch

stretch_to_branch:
    JMP     branch
```

Defines:
`do_chk`, used in chunk 176b.
`instr_inc_chk`, used in chunk 109.
`stretch_to_branch`, used in chunks 179–81.

Uses `MOVW` 12a, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, `branch` 161a, `cmp16` 101b, `inc_var` 160a, and `negated_branch` 161a.

14.7.3 je

`je` jumps if the first operand is equal to any of the next operands. However, in negative node (`jne`), we jump if the first operand is not equal to any of the next operands.

First, we check that there is at least one operand, and if not, we hit a `BRK`.

177b ⟨*Instruction je* 177b⟩≡ (207) 178a ▷

```
instr_je:
    SUBROUTINE

    LDX     OPERAND_COUNT
    DEX
    BNE     .check_second
    JSR     brk
```

Defines:
`instr_je`, used in chunk 109.

Uses `OPERAND_COUNT` 204 and `brk` 31c.

Next, we check against the second operand, and if it's equal, we branch, and if that was the last operand, we negative branch.

178a ⟨*Instruction je* 177b⟩+≡ (207) ◁177b 178b▷
```
  .check_second:
      LDA       OPERAND0
      CMP       OPERAND1
      BNE       .check_next
      LDA       OPERAND0+1
      CMP       OPERAND1+1
      BEQ       .branch

  .check_next:
      DEX
      BEQ       .neg_branch
```
Uses `OPERAND0` 204, `OPERAND1` 204, and `branch` 161a.

Next we do the same with the third operand.

178b ⟨*Instruction je* 177b⟩+≡ (207) ◁178a 178c▷
```
      LDA       OPERAND0
      CMP       OPERAND0+4
      BNE       .check_next2
      LDA       OPERAND0+1
      CMP       OPERAND0+5
      BEQ       .branch

  .check_next2:
      DEX
      BEQ       .neg_branch
```
Uses `OPERAND0` 204 and `branch` 161a.

And again with the fourth operand.

178c ⟨*Instruction je* 177b⟩+≡ (207) ◁178b
```
      LDA       OPERAND0
      CMP       OPERAND0+6
      BNE       .check_second      ; why not just go to .neg_branch?
      LDA       OPERAND0+1
      CMP       OPERAND0+7
      BEQ       .branch

  .neg_branch:
      JMP       negated_branch

  .branch:
      JMP       branch
```
Uses `OPERAND0` 204, `branch` 161a, and `negated_branch` 161a.

14.7.4 jg

`jg` jumps if the first operand is greater than the second operand, in a signed comparison. In negative mode (`jle`), we jump if the first operand is less than or equal to the second operand.

179a ⟨*Instruction jg* 179a⟩≡ (207)

```
instr_jg:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH1
    MOVW     OPERAND1, SCRATCH2
    JSR      cmp16
    BCC      stretch_to_branch
    JMP      negated_branch
```

Defines:
`instr_jg`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, `cmp16` 101b, `negated_branch` 161a, and `stretch_to_branch` 177a.

14.7.5 jin

`jin` jumps if the first operand is a child object of the second operand.

179b ⟨*Instruction jin* 179b⟩≡ (207)

```
instr_jin:
    SUBROUTINE

    LDA      OPERAND0
    JSR      get_object_addr
    LDY      #OBJECT_PARENT_OFFSET
    LDA      OPERAND1
    CMP      (SCRATCH2),Y
    BEQ      stretch_to_branch
    JMP      negated_branch
```

Defines:
`instr_jin`, used in chunk 109.

Uses `OBJECT_PARENT_OFFSET` 206a, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH2` 204, `get_object_addr` 132, `negated_branch` 161a, and `stretch_to_branch` 177a.

14.7.6 jl

`jl` jumps if the first operand is less than the second operand, in a signed comparison. In negative mode (`jge`), we jump if the first operand is greater than or equal to the second operand.

180a ⟨*Instruction jl* 180a⟩≡ (207)
```
  instr_jl:
      SUBROUTINE

      MOVW     OPERAND0, SCRATCH2
      MOVW     OPERAND1, SCRATCH1
      JSR      cmp16
      BCC      stretch_to_branch
      JMP      negated_branch
```
Defines:
`instr_jl`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, `cmp16` 101b, `negated_branch` 161a, and `stretch_to_branch` 177a.

14.7.7 jz

`jz` jumps if its operand is `0`.

This also includes a "stretchy jump" for other instructions that need to branch.

180b ⟨*Instruction jz* 180b⟩≡ (207)
```
  instr_jz:
      SUBROUTINE

      LDA      OPERAND0+1
      ORA      OPERAND0
      BEQ      take_branch
      JMP      negated_branch

  take_branch:
      JMP      branch
```
Defines:
`instr_jz`, used in chunk 109.
`take_branch`, used in chunk 188.

Uses `OPERAND0` 204, `branch` 161a, and `negated_branch` 161a.

14.7.8 test

`test` jumps if all the bits in the first operand are set in the second operand.

181a ⟨*Instruction test* 181a⟩≡ (207)

```
instr_test:
    SUBROUTINE

    MOVB    OPERAND1+1, SCRATCH2+1
    AND     OPERAND0+1
    STA     SCRATCH1+1
    MOVB    OPERAND1, SCRATCH2
    AND     OPERAND0
    STA     SCRATCH1
    JSR     cmpu16
    BEQ     stretch_to_branch
    JMP     negated_branch
```

Defines:
  `instr_test`, used in chunk 109.
Uses `MOVB` 11b, `OPERAND0` 204, `OPERAND1` 204, `SCRATCH1` 204, `SCRATCH2` 204, `cmpu16` 101a, `negated_branch` 161a, and `stretch_to_branch` 177a.

14.7.9 test_attr

`test_attr` jumps if the object in the first operand has the attribute number in the second operand set. This is done by getting the attribute word and mask for the attribute number, and then bitwise-anding them together. If the result is nonzero, the attribute is set.

181b ⟨*Instruction test attr* 181b⟩≡ (207)

```
instr_test_attr:
    SUBROUTINE

    JSR     attr_ptr_and_mask
    LDA     SCRATCH1+1
    AND     SCRATCH3+1
    STA     SCRATCH1+1
    LDA     SCRATCH1
    AND     SCRATCH3
    ORA     SCRATCH1+1
    BNE     stretch_to_branch
    JMP     negated_branch
```

Defines:
  `instr_test_attr`, used in chunk 109.
Uses `SCRATCH1` 204, `SCRATCH3` 204, `attr_ptr_and_mask` 138, `negated_branch` 161a, and `stretch_to_branch` 177a.

14.8 Jump and subroutine instructions

14.8.1 call

`call` calls the routine at the given address.This instruction has been described in Call.

14.8.2 jump

`jump` jumps relative to the signed operand. We subtract `1` from the operand so that we can call `branch_to_offset`, which does another decrement. Thus, the address to go to is the address after this instruction, plus the operand, minus 2.

182a ⟨*Instruction jump* 182a⟩≡ (207)

```
instr_jump:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2
    SUBB    SCRATCH2, #$01
    JMP     branch_to_offset
```

Defines:
`instr_jump`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `SCRATCH2` 204, `SUBB` 16b, and `branch_to_offset` 163b.

14.8.3 print_ret

`print_ret` is the same as `print`, except that it prints a `CRLF` after the string, and then calls the `rtrue` instruction.

182b ⟨*Instruction print ret* 182b⟩≡ (207)

```
instr_print_ret:
    SUBROUTINE

    JSR     print_string_literal
    LDA     #$0D
    JSR     buffer_char
    LDA     #$0A
    JSR     buffer_char
    JMP     instr_rtrue
```

Defines:
`instr_print_ret`, used in chunk 109.

Uses `buffer_char` 57 and `instr_rtrue` 184a.

14.8.4 ret

`ret` returns from a routine. The operand is the return value. This instruction has been described in Return.

14.8.5 ret_popped

`ret_popped` pops the stack and returns that value.

183a ⟨*Instruction ret popped* 183a⟩≡ (207)

```
  instr_ret_popped:
      SUBROUTINE

      JSR     pop
      MOVW    SCRATCH2, OPERAND0
      JMP     instr_ret
```

Defines:
  `instr_ret_popped`, used in chunk 109.
Uses `MOVW` 12a, `OPERAND0` 204, `SCRATCH2` 204, `instr_ret` 129, and `pop` 36.

14.8.6 rfalse

`rfalse` places `#$0000` into `OPERAND0`, and then calls the `ret` instruction.

183b ⟨*Instruction rfalse* 183b⟩≡ (207)

```
  instr_rfalse:
      SUBROUTINE

      LDA     #$00
      JMP     ret_a
```

Defines:
  `instr_rfalse`, used in chunks 109 and 163a.
Uses `ret_a` 184a.

14.8.7 rtrue

`rtrue` places `#$0001` into `OPERAND0`, and then calls the `ret` instruction.

184a ⟨*Instruction rtrue* 184a⟩≡ (207)
```
instr_rtrue:
    SUBROUTINE

    LDA     #$01
ret_a:
    STA     OPERAND0
    LDA     #$00
    STA     OPERAND0+1
    JMP     instr_ret
```
Defines:
  `instr_rtrue`, used in chunks 109, 163a, and 182b.
  `ret_a`, used in chunk 183b.
Uses `OPERAND0` 204 and `instr_ret` 129.

14.9 Print instructions

14.9.1 new_line

`new_line` prints `CRLF`.

184b ⟨*Instruction new line* 184b⟩≡ (207)
```
instr_new_line:
    SUBROUTINE

    LDA     #$0D
    JSR     buffer_char
    LDA     #$0A
    JSR     buffer_char
    JMP     do_instruction
```
Defines:
  `instr_new_line`, used in chunk 109.
Uses `buffer_char` 57 and `do_instruction` 112.

14.9.2 print

`print` treats the following bytes of z-code as a z-encoded string, and prints it to the output.

185a ⟨*Instruction print* 185a⟩≡ (207)

```
instr_print:
    SUBROUTINE

    JSR     print_string_literal
    JMP     do_instruction
```

Defines:
  `instr_print`, used in chunk 109.
Uses `do_instruction` 112.

14.9.3 print_addr

`print_addr` prints the z-encoded string at the address given by the operand.

185b ⟨*Instruction print addr* 185b⟩≡ (207)

```
instr_print_addr:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2
    JSR     load_address
    JMP     print_zstring_and_next
```

Defines:
  `instr_print_addr`, used in chunk 109.
Uses `MOVW` 12a, `OPERAND0` 204, `SCRATCH2` 204, `load_address` 45b, and `print_zstring_and_next` 159b.

14.9.4 print_char

`print_char` prints the one-byte ASCII character in `OPERAND0`.

185c ⟨*Instruction print char* 185c⟩≡ (207)

```
instr_print_char:
    SUBROUTINE

    LDA     OPERAND0
    JSR     buffer_char
    JMP     do_instruction
```

Defines:
  `instr_print_char`, used in chunk 109.
Uses `OPERAND0` 204, `buffer_char` 57, and `do_instruction` 112.

14.9.5 print_num

`print_num` prints the 16-bit signed value in `OPERAND0` as a decimal number.

186a ⟨*Instruction print num* 186a⟩≡ (207)

```
instr_print_num:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2
    JSR     print_number
    JMP     do_instruction
```

Defines:
`instr_print_num`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `SCRATCH2` 204, `do_instruction` 112, and `print_number` 103.

14.9.6 print_obj

`print_obj` prints the short name of the object in the operand.

186b ⟨*Instruction print obj* 186b⟩≡ (207)

```
instr_print_obj:
    SUBROUTINE

    LDA     OPERAND0
    JSR     print_obj_in_A
    JMP     do_instruction
```

Defines:
`instr_print_obj`, used in chunk 109.

Uses `OPERAND0` 204, `do_instruction` 112, and `print_obj_in_A` 137.

14.9.7 print_paddr

`print_paddr` prints the z-encoded string at the packed address in the operand.

186c ⟨*Instruction print paddr* 186c⟩≡ (207)

```
instr_print_paddr:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2      ; Z_PC2 <- OPERAND0 * 2
    JSR     load_packed_address

    ; Falls through to print_zstring_and_next
```

Defines:
`instr_print_paddr`, used in chunk 109.

Uses `MOVW` 12a, `OPERAND0` 204, `SCRATCH2` 204, `load_packed_address` 46, and `print_zstring_and_next` 159b.

14.10 Object instructions

14.10.1 clear_attr

`clear_attr` clears the attribute number in the second operand for the object in the first operand. This is done by getting the attribute word and mask for the attribute number, and then bitwise-anding the inverse of the mask with the attribute word, and storing the result.

187 ⟨*Instruction clear attr* 187⟩≡ (207)

```
instr_clear_attr:
    SUBROUTINE

    JSR     attr_ptr_and_mask
    LDY     #$01
    LDA     SCRATCH3
    EOR     #$FF
    AND     SCRATCH1
    STA     (SCRATCH2),Y
    DEY
    LDA     SCRATCH3+1
    EOR     #$FF
    AND     SCRATCH1+1
    STA     (SCRATCH2),Y
    JMP     do_instruction
```

Defines:
`instr_clear_attr`, used in chunk 109.

Uses `SCRATCH1` 204, `SCRATCH2` 204, `SCRATCH3` 204, `attr_ptr_and_mask` 138, and `do_instruction` 112.

14.10.2 get_child

`get_child` gets the first child object of the object in the operand, stores it into the variable in the next code byte, and branches if it exists (i.e. is not `0`).

188 ⟨*Instruction get child* 188⟩≡ (207)

```
instr_get_child:
    LDA      OPERAND0
    JSR      get_object_addr
    LDY      #OBJECT_CHILD_OFFSET

push_and_check_obj:
    LDA      (SCRATCH2),Y
    PHA
    STA      SCRATCH2
    LDA      #$00
    STA      SCRATCH2+1
    JSR      store_var     ; store in var of next code byte.
    PLA
    ORA      #$00
    BNE      take_branch
    JMP      negated_branch
```

Defines:
`push_and_check_obj`, used in chunk 196.

Uses `OBJECT_CHILD_OFFSET` 206a, `OPERAND0` 204, `SCRATCH2` 204, `get_object_addr` 132, `negated_branch` 161a, `store_var` 123, and `take_branch` 180b.

14.10.3 get_next_prop

`get_next_prop` gets the next property number for the object in the first operand after the property number in the second operand, and stores it in the variable in the next code byte. If there is no next property, zero is stored.

If the property number in the second operand is zero, the first property number of the object is returned.

189 ⟨*Instruction get next prop* 189⟩≡ (207)

```
instr_get_next_prop:
    SUBROUTINE

    JSR     get_property_ptr
    LDA     OPERAND1
    BEQ     .store

.loop:
    JSR     get_property_num
    CMP     OPERAND1
    BEQ     .found
    BCS     .continue
    JMP     store_zero_and_next

.continue:
    JSR     next_property
    JMP     .loop

.store:
    JSR     get_property_num
    JMP     store_A_and_next

.found:
    JSR     next_property
    JMP     .store
```

Defines:
  `instr_get_next_prop`, used in chunk 109.

Uses `OPERAND1` 204, `get_property_num` 141a, `get_property_ptr` 140, `next_property` 142, `store_A_and_next` 159a, and `store_zero_and_next` 159a.

14.10.4 get_parent

`get_parent` gets the parent object of the object in the operand, and stores it into the variable in the next code byte.

190 ⟨*Instruction get parent* 190⟩≡ (207)

```
instr_get_parent:
    SUBROUTINE

    LDA     OPERAND0
    JSR     get_object_addr
    LDY     #OBJECT_PARENT_OFFSET
    LDA     (SCRATCH2),Y
    STA     SCRATCH2
    LDA     #$00
    STA     SCRATCH2+1
    JSR     store_and_next
```

Defines:
`instr_get_parent`, used in chunk 109.

Uses `OBJECT_PARENT_OFFSET` 206a, `OPERAND0` 204, `SCRATCH2` 204, `get_object_addr` 132, and `store_and_next` 159a.

14.10.5 get_prop

`get_prop` gets the property number in the second operand for the object in the first operand, and stores the value of the property in the variable in the next code byte. If the object doesn't have the property, the default value for the property is used. If the property length is 1, then the byte is zero-extended and stored. If the property length is 2, then the entire word is stored. If the property length is anything else, we hit a `BRK`.

First, we check to see if the property is in the object's properties.

191 ⟨*Instruction get prop* 191⟩≡ (207) 192▷

```
instr_get_prop:
    SUBROUTINE

    JSR     get_property_ptr

.loop:
    JSR     get_property_num
    CMP     OPERAND1
    BEQ     .found
    BCC     .get_default
    JSR     next_property
    JMP     .loop
```

Defines:
`instr_get_prop`, used in chunk 109.

Uses `OPERAND1` 204, `get_property_num` 141a, `get_property_ptr` 140, and `next_property` 142.

To get the default value, we look in the beginning of the object table, and index into the word containing the property default. Then we store it and we're done.

192 ⟨*Instruction get prop* 191⟩+≡ (207) ◁191 193▷

```
.get_default:
    LDY      #HEADER_OBJECT_TABLE_ADDR_OFFSET
    CLC
    LDA      (Z_HEADER_ADDR),Y
    ADC      Z_HEADER_ADDR
    STA      SCRATCH1
    DEY
    LDA      (Z_HEADER_ADDR),Y
    ADC      Z_HEADER_ADDR+1
    STA      SCRATCH1+1              ; table_ptr
    LDA      OPERAND1                ; SCRATCH2 <- table_ptr[2*OPERAND1]
    ASL
    TAY
    DEY
    LDA      (SCRATCH1),Y
    STA      SCRATCH2
    DEY
    LDA      (SCRATCH1),Y
    STA      SCRATCH2+1
    JMP      store_and_next
```

Uses HEADER_OBJECT_TABLE_ADDR_OFFSET 206a, OPERAND1 204, SCRATCH1 204, SCRATCH2 204, and store_and_next 159a.

If the property was found, we load the zero-extended byte or the word, depending on the property length. Also if the property length is not valid, we hit a BRK.

193 ⟨*Instruction get prop* 191⟩+≡ (207) ◁192

```
.found:
    JSR     get_property_len
    INY
    CMP     #$00
    BEQ     .byte_prop
    CMP     #$01
    BEQ     .word_prop
    JSR     brk

.word_prop:
    LDA     (SCRATCH2),Y
    STA     SCRATCH1+1
    INY
    LDA     (SCRATCH2),Y
    STA     SCRATCH1
    MOVW    SCRATCH1, SCRATCH2
    JMP     store_and_next

.byte_prop:
    LDA     (SCRATCH2),Y
    STA     SCRATCH2
    LDA     #$00
    STA     SCRATCH2+1
    JMP     store_and_next
```

Uses `MOVW` 12a, `SCRATCH1` 204, `SCRATCH2` 204, `brk` 31c, `get_property_len` 141b, and `store_and_next` 159a.

14.10.6 get_prop_addr

`get_prop_addr` gets the Z-address of the property number in the second operand for the object in the first operand, and stores it in the variable in the next code byte. If the object does not have the property, zero is stored.

194 ⟨*Instruction get prop addr* 194⟩≡ (207)

```
instr_get_prop_addr:
    SUBROUTINE

    JSR      get_property_ptr

.loop:
    JSR      get_property_num
    CMP      OPERAND1
    BEQ      .found
    BCS      .next
    JMP      store_zero_and_next

.next:
    JSR      next_property
    JMP      .loop

.found:
    INCW     SCRATCH2
    CLC
    TYA
    ADDAC    SCRATCH2
    SUBW     SCRATCH2, Z_HEADER_ADDR, SCRATCH2
    JMP      store_and_next
```

Defines:
`instr_get_prop_addr`, used in chunk 109.

Uses `ADDAC` 14b, `INCW` 13b, `OPERAND1` 204, `SCRATCH2` 204, `SUBW` 17b, `get_property_num` 141a, `get_property_ptr` 140, `next_property` 142, `store_and_next` 159a, and `store_zero_and_next` 159a.

14.10.7 get_prop_len

get_prop_len gets the length of the property data for the property address in the operand, and stores it into the variable in the next code byte. The address in the operand is relative to the start of the header, and points to the property data. The property's one-byte length is stored at that address minus one.

195 ⟨*Instruction get prop len* 195⟩≡ (207)

```
instr_get_prop_len:
    CLC
    LDA         OPERAND0
    ADC         Z_HEADER_ADDR
    STA         SCRATCH2
    LDA         OPERAND0+1
    ADC         Z_HEADER_ADDR+1
    STA         SCRATCH2+1
    LDA         SCRATCH2
    SEC
    SBC         #$01
    STA         SCRATCH2
    BCS         .continue
    DEC         SCRATCH2+1

.continue:
    LDY         #$00
    JSR         get_property_len
    CLC
    ADC         #$01
    JMP         store_A_and_next
```

Defines:
instr_get_prop_len, used in chunk 109.

Uses OPERAND0 204, SCRATCH2 204, get_property_len 141b, and store_A_and_next 159a.

14.10.8 get_sibling

`get_sibling` gets the next object of the object in the operand (its "sibling"), stores it into the variable in the next code byte, and branches if it exists (i.e. is not `0`).

196 ⟨*Instruction get sibling* 196⟩≡ (207)

```
  instr_get_sibling:
      SUBROUTINE

      LDA      OPERAND0
      JSR      get_object_addr
      LDY      #OBJECT_SIBLING_OFFSET
      JMP      push_and_check_obj
```

Defines:
  `instr_get_sibling`, used in chunk 109.

Uses `OBJECT_SIBLING_OFFSET` 206a, `OPERAND0` 204, `get_object_addr` 132,
  and `push_and_check_obj` 188.

14.10.9 insert_obj

insert_obj inserts the object in OPERAND0 as a child of the object in OPERAND1}. It becomes the first child in the object.

197 ⟨*Instruction insert obj* 197⟩≡ (207)

```
  instr_insert_obj:
      JSR      remove_obj             ; remove_obj<OPERAND0>
      LDA      OPERAND0
      JSR      get_object_addr        ; obj_ptr = get_object_addr<OPERAND0>
      PSHW     SCRATCH2
      LDY      #OBJECT_PARENT_OFFSET
      LDA      OPERAND1
      STA      (SCRATCH2),Y           ; obj_ptr->parent = OPERAND1
      JSR      get_object_addr        ; dest_ptr = get_object_addr<OPERAND1>
      LDY      #OBJECT_CHILD_OFFSET   ; tmp = dest_ptr->child
      LDA      (SCRATCH2),Y
      TAX
      LDA      OPERAND0               ; dest_ptr->child = OPERAND0
      STA      (SCRATCH2),Y
      PULW     SCRATCH2
      TXA
      BEQ      .continue
      LDY      #OBJECT_SIBLING_OFFSET ; obj_ptr->sibling = tmp
      STA      (SCRATCH2),Y

  .continue:
      JMP      do_instruction
```

Defines:
instr_insert_obj, used in chunk 109.

Uses OBJECT_CHILD_OFFSET 206a, OBJECT_PARENT_OFFSET 206a, OBJECT_SIBLING_OFFSET 206a, OPERAND0 204, OPERAND1 204, PSHW 12b, PULW 13a, SCRATCH2 204, do_instruction 112, get_object_addr 132, and remove_obj 134a.

14.10.10 put_prop

put_prop stores the value in OPERAND2 into property number OPERAND1 in object OPERAND0. The property must exist, and must be of length 1 or 2, otherwise a BRK is hit.

198 ⟨*Instruction put prop* 198⟩≡ (207)

```
instr_put_prop:
    SUBROUTINE

    JSR     get_property_ptr

.loop:
    JSR     get_property_num
    CMP     OPERAND1
    BEQ     .found
    BCS     .continue
    JSR     brk

.continue:
    JSR     next_property
    JMP     .loop

.found:
    JSR     get_property_len
    INY
    CMP     #$00
    BEQ     .byte_property
    CMP     #$01
    BEQ     .word_property
    JSR     brk

.word_property:
    LDA     OPERAND2+1
    STA     (SCRATCH2),Y
    INY
    LDA     OPERAND2
    STA     (SCRATCH2),Y
    JMP     do_instruction

.byte_property:
    LDA     OPERAND2
    STA     (SCRATCH2),Y
    JMP     do_instruction
```

Defines:
instr_put_prop, used in chunk 109.

Uses OPERAND1 204, OPERAND2 204, SCRATCH2 204, brk 31c, do_instruction 112, get_property_len 141b, get_property_num 141a, get_property_ptr 140, and next_property 142.

14.10.11 remove_obj

`remove_obj` removes the object in the operand from the object tree.

199a ⟨*Instruction remove obj* 199a⟩≡ (207)

```
instr_remove_obj:
    SUBROUTINE

    JSR     remove_obj
    JMP     do_instruction
```

Defines:
`instr_remove_obj`, used in chunk 109.

Uses `do_instruction` 112 and `remove_obj` 134a.

14.10.12 set_attr

`set_attr` sets the attribute number in the second operand for the object in the first operand. This is done by getting the attribute word and mask for the attribute number, and then bitwise-oring them together, and storing the result.

199b ⟨*Instruction set attr* 199b⟩≡ (207)

```
instr_set_attr:
    SUBROUTINE

    JSR     attr_ptr_and_mask
    LDY     #$01
    LDA     SCRATCH1
    ORA     SCRATCH3
    STA     (SCRATCH2),Y
    DEY
    LDA     SCRATCH1+1
    ORA     SCRATCH3+1
    STA     (SCRATCH2),Y
    JMP     do_instruction
```

Defines:
`instr_set_attr`, used in chunk 109.

Uses `SCRATCH1` 204, `SCRATCH2` 204, `SCRATCH3` 204, `attr_ptr_and_mask` 138, and `do_instruction` 112.

14.11 Other instructions

14.11.1 nop

`nop` does nothing.

200a ⟨*Instruction nop* 200a⟩≡ (207)

```
instr_nop:
    SUBROUTINE

    JMP       do_instruction
```

Defines:
`instr_nop`, used in chunk 109.
Uses `do_instruction` 112.

14.11.2 restart

`restart` restarts the game. This dumps the buffer, and then jumps back to `main`.

200b ⟨*Instruction restart* 200b⟩≡ (207)

```
instr_restart:
    SUBROUTINE

    JSR       dump_buffer_with_more
    JMP       main
```

Defines:
`instr_restart`, used in chunk 109.
Uses `dump_buffer_with_more` 54 and `main` 26a.

14.11.3 restore

`restore` restores the game. See the section Restoring the game state.

14.11.4 quit

`quit` quits the game by printing "– END OF SESSION –" and then spinlooping.

201 ⟨*Instruction quit* 201⟩≡ (207)
```
sEndOfSession:
    DC       "-- END OF SESSION --"

instr_quit:
    SUBROUTINE

    JSR      dump_buffer_with_more
    STOW     sEndOfSession, SCRATCH2
    LDX      #20
    JSR      print_ascii_string
    JSR      dump_buffer_with_more

.spinloop:
    JMP      .spinloop
```
Defines:
`instr_quit`, used in chunk 109.
Uses `SCRATCH2` 204, `STOW` 10, `dump_buffer_with_more` 54, and `print_ascii_string` 59b.

14.11.5 save

`save` saves the game. See the section Saving the game state.

14.11.6 sread

`sread` reads a line of input from the keyboard and parses it. See the section Lexical parsing.

Chapter 15

The entire program

202a ⟨*main.asm* 202a⟩≡
```
        PROCESSOR 6502
```
⟨*Macros* 10⟩
⟨*defines* 202b⟩
⟨*routines* 207⟩

202b ⟨*defines* 202b⟩≡ (202a)
⟨*Apple ROM defines* 203⟩
⟨*Program defines* 204⟩
⟨*Table offsets* 206a⟩
⟨*variable numbers* 206b⟩

203 ⟨*Apple ROM defines* 203⟩≡ (202b)

```
WNDLFT      EQU     $20
WNDWDTH     EQU     $21
WNDTOP      EQU     $22
WNDBTM      EQU     $23
CH          EQU     $24
CV          EQU     $25
IWMDATAPTR  EQU     $26     ; IWM pointer to write disk data to
IWMSLTNDX   EQU     $2B     ; IWM Slot times 16
INVFLG      EQU     $32
PROMPT      EQU     $33
CSW         EQU     $36     ; 2 bytes

; Details https://6502disassembly.com/a2-rom/APPLE2.ROM.html
IWMSECTOR   EQU     $3D  ; IWM sector to read
RDSECT_PTR  EQU     $3E  ; 2 bytes
RANDOM_VAL  EQU     $4E  ; 2 bytes

INIT        EQU     $FB2F
VTAB        EQU     $FC22
HOME        EQU     $FC58
CLREOL      EQU     $FC9C
RDKEY       EQU     $FD0C
GETLN1      EQU     $FD6F
COUT        EQU     $FDED
COUT1       EQU     $FDF0
SETVID      EQU     $FE93
SETKBD      EQU     $FE89
```

Defines:
- CH, used in chunks 54, 69, and 145.
- CLREOL, used in chunks 54, 69, and 145.
- COUT, used in chunks 51 and 55b.
- COUT1, used in chunks 47, 50, and 55a.
- CSW, used in chunks 51 and 55b.
- CV, used in chunk 69.
- GETLN1, used in chunk 71.
- HOME, used in chunk 48.
- INIT, used in chunks 23 and 24.
- INVFLG, used in chunks 49, 54, 69, and 145.
- IWMDATAPTR, used in chunks 21a, 22e, and 24.
- IWMSECTOR, used in chunks 22c and 24.
- IWMSLTNDX, used in chunks 21–24.
- PROMPT, used in chunk 49.
- RDKEY, used in chunks 54, 145, 147a, and 148.
- RDSECT_PTR, used in chunks 20c, 21b, and 24.
- SETKBD, used in chunks 23 and 24.
- SETVID, used in chunks 23 and 24.
- VTAB, used in chunk 69.
- WNDBTM, used in chunks 49 and 54.
- WNDLFT, used in chunk 49.
- WNDTOP, used in chunks 48, 49, 54, and 71.
- WNDWDTH, used in chunks 49, 55a, and 57–59.

204 ⟨*Program defines 204*⟩≡ (202b)

```
DEBUG_JUMP          EQU     $7C      ; 3 bytes
SECTORS_PER_TRACK   EQU     $7F
CURR_OPCODE         EQU     $80
OPERAND_COUNT       EQU     $81
OPERAND0            EQU     $82      ; 2 bytes
OPERAND1            EQU     $84      ; 2 bytes
OPERAND2            EQU     $86      ; 2 bytes
OPERAND3            EQU     $88      ; 2 bytes
Z_PC                EQU     $8A      ; 3 bytes
ZCODE_PAGE_ADDR     EQU     $8D      ; 2 bytes
ZCODE_PAGE_VALID    EQU     $8F
PAGE_TABLE_INDEX    EQU     $90
Z_PC2_H             EQU     $91
Z_PC2_HH            EQU     $92
Z_PC2_L             EQU     $93
ZCODE_PAGE_ADDR2    EQU     $94      ; 2 bytes
ZCODE_PAGE_VALID2   EQU     $96
PAGE_TABLE_INDEX2   EQU     $97
GLOBAL_ZVARS_ADDR   EQU     $98      ; 2 bytes
LOCAL_ZVARS         EQU     $9A      ; 30 bytes
AFTER_Z_IMAGE_ADDR  EQU     $B8
Z_HEADER_ADDR       EQU     $BA      ; 2 bytes
NUM_IMAGE_PAGES     EQU     $BC
FIRST_Z_PAGE        EQU     $BD
LAST_Z_PAGE         EQU     $BF
PAGE_L_TABLE        EQU     $C0      ; 2 bytes
PAGE_H_TABLE        EQU     $C2      ; 2 bytes
NEXT_PAGE_TABLE     EQU     $C4      ; 2 bytes
PREV_PAGE_TABLE     EQU     $C6      ; 2 bytes
STACK_COUNT         EQU     $C8
Z_SP                EQU     $C9      ; 2 bytes
FRAME_Z_SP          EQU     $CB      ; 2 bytes
FRAME_STACK_COUNT   EQU     $CD
SHIFT_ALPHABET      EQU     $CE
LOCKED_ALPHABET     EQU     $CF
ZDECOMPRESS_STATE   EQU     $D0
ZCHARS_L            EQU     $D1
ZCHARS_H            EQU     $D2
ZCHAR_SCRATCH1      EQU     $D3      ; 6 bytes
ZCHAR_SCRATCH2      EQU     $DA      ; 6 bytes
TOKEN_IDX           EQU     $E0
INPUT_PTR           EQU     $E1
Z_ABBREV_TABLE      EQU     $E2      ; 2 bytes
SCRATCH1            EQU     $E4      ; 2 bytes
SCRATCH2            EQU     $E6      ; 2 bytes
SCRATCH3            EQU     $E8      ; 2 bytes
SIGN_BIT            EQU     $EA
BUFF_END            EQU     $EB
BUFF_LINE_LEN       EQU     $EC
```

```
CURR_LINE           EQU     $ED
PRINTER_CSW         EQU     $EE     ; 2 bytes
TMP_Z_PC            EQU     $F0     ; 3 bytes
BUFF_AREA           EQU     $0200
RWTS                EQU     $2900
```

Defines:

- AFTER_Z_IMAGE_ADDR, used in chunks 33a, 40, and 43.
- BUFF_AREA, used in chunks 50, 51, 57–59, 71, 107, 108, 149b, 150a, 152c, and 153c.
- BUFF_END, used in chunks 50, 51, 55a, 57–59, and 71.
- BUFF_LINE_LEN, used in chunks 58b and 59a.
- CURR_DISK_BUFF_ADDR, never used.
- CURR_LINE, used in chunks 48, 54, and 71.
- CURR_OPCODE, used in chunks 112, 115–17, and 119.
- DEBUG_JUMP, used in chunk 111.
- FIRST_Z_PAGE, used in chunks 28b, 33b, 41, and 42.
- FRAME_STACK_COUNT, used in chunks 126a and 128–30.
- FRAME_Z_SP, used in chunks 126a and 128–30.
- GLOBAL_ZVARS_ADDR, used in chunks 32, 121, and 123.
- LAST_Z_PAGE, used in chunks 28b, 33b, 41, and 42.
- LOCAL_ZVARS, used in chunks 121, 123, 127, 128a, 130b, 150b, and 153b.
- LOCKED_ALPHABET, used in chunks 60, 62, 64, 65, 81, 82b, 84a, and 86.
- NEXT_PAGE_TABLE, used in chunks 27, 28a, 33b, and 42.
- NUM_IMAGE_PAGES, used in chunks 30, 33a, 38, and 43.
- OPERAND0, used in chunks 71, 73, 75b, 76a, 79, 114d, 116a, 118, 125, 126b, 128a, 131, 134–36, 138, 140, 160, 165–76, 178–86, 188, 190, and 195–97.
- OPERAND1, used in chunks 73–75, 77, 78, 116b, 138, 165–68, 170–81, 189, 191, 192, 194, 197, and 198.
- OPERAND2, used in chunks 167, 168a, and 198.
- OPERAND3, never used.
- OPERAND_COUNT, used in chunks 112, 114d, 117a, 118, 128a, and 177b.
- PAGE_H_TABLE, used in chunks 27, 28a, 40, 41, and 43.
- PAGE_L_TABLE, used in chunks 27, 28a, 40, 41, and 43.
- PAGE_TABLE_INDEX, used in chunks 38, 40, and 43.
- PAGE_TABLE_INDEX2, used in chunks 40 and 43.
- PREV_PAGE_TABLE, used in chunks 27, 28a, and 42.
- PRINTER_CSW, used in chunks 27, 51, and 55b.
- RWTS, used in chunk 106.
- SCRATCH1, used in chunks 29b, 30, 40, 43, 78, 81–84, 86, 87, 89, 91–94, 96a, 97, 100, 101, 103, 106–108, 111, 121, 123, 128a, 130, 135–40, 147b, 163c, 164, 171–74, 176b, 177a, 179–81, 187, 192, 193, and 199b.
- SCRATCH2, used in chunks 29b, 30, 34–36, 38, 40–43, 45–47, 54, 59b, 61, 65, 69, 80–83, 90–97, 100, 101, 103, 106–108, 111, 113–21, 123–28, 130–41, 143, 145, 147–51, 153, 154, 159, 160, 162–77, 179–83, 185–88, 190, 192–95, 197–99, and 201.
- SCRATCH3, used in chunks 59b, 63a, 64, 66–68, 73–79, 81, 82, 84c, 86–89, 91–93, 97, 100, 103, 128a, 130, 139a, 151b, 154b, 181b, 187, and 199b.
- SECTORS_PER_TRACK, used in chunk 106.
- SHIFT_ALPHABET, used in chunks 60, 62, 64, and 65.
- STACK_COUNT, used in chunks 27, 35, 36, 128b, 129, 150b, and 153b.
- TMP_Z_PC, used in chunk 112.
- ZCHARS_H, used in chunks 61 and 65.
- ZCHARS_L, used in chunks 61 and 65.
- ZCHAR_SCRATCH1, used in chunks 27, 75, 76, 82a, and 83b.
- ZCHAR_SCRATCH2, used in chunks 81, 84–87, 89, 92a, and 93.
- ZCODE_PAGE_ADDR, used in chunks 37, 39, and 68b.
- ZCODE_PAGE_ADDR2, used in chunks 43 and 68b.
- ZCODE_PAGE_VALID, used in chunks 27, 37, 39, 43, 68b, 126a, 131, 153b, and 164.
- ZCODE_PAGE_VALID2, used in chunks 27, 40, 43, 46, 65, and 68b.

ZDECOMPRESS_STATE, used in chunks 61, 62, and 65.
Z_ABBREV_TABLE, used in chunks 32 and 65.
Z_PC, used in chunks 31d, 37, 38, 40, 68b, 112, 121, 126, 130d, 149c, 153b, and 164.
Z_PC2_H, used in chunks 43, 45b, 46, 65, and 68b.
Z_PC2_HH, used in chunks 43, 45b, 46, 65, and 68b.
Z_PC2_L, used in chunks 43, 45b, 46, 65, and 68b.
Z_SP, used in chunks 27, 35, 36, 128b, and 129.

206a ⟨*Table offsets* 206a⟩≡ (202b)

```
HEADER_DICT_OFFSET        EQU     $08
HEADER_OBJECT_TABLE_ADDR_OFFSET  EQU     $0B
HEADER_STATIC_MEM_BASE    EQU     $0E
HEADER_FLAGS2_OFFSET      EQU     $10
FIRST_OBJECT_OFFSET       EQU     $35

OBJECT_PARENT_OFFSET      EQU     $04
OBJECT_SIBLING_OFFSET     EQU     $05
OBJECT_CHILD_OFFSET       EQU     $06
OBJECT_PROPS_OFFSET       EQU     $07
```

Defines:
FIRST_OBJECT_OFFSET, used in chunk 133a.
HEADER_DICT_OFFSET, used in chunk 90.
HEADER_FLAGS2_OFFSET, used in chunk 71.
HEADER_OBJECT_TABLE_ADDR_OFFSET, used in chunks 133b and 192.
HEADER_STATIC_MEM_BASE, used in chunk 151b.
OBJECT_CHILD_OFFSET, used in chunks 134c, 135b, 188, and 197.
OBJECT_PARENT_OFFSET, used in chunks 134a, 135c, 179b, 190, and 197.
OBJECT_PROPS_OFFSET, used in chunks 137 and 140.
OBJECT_SIBLING_OFFSET, used in chunks 135b, 136a, 196, and 197.

206b ⟨*variable numbers* 206b⟩≡ (202b)

```
VAR_CURR_ROOM        EQU     $10
VAR_SCORE            EQU     $11
VAR_MAX_SCORE        EQU     $12
```

Defines:
VAR_CURR_ROOM, used in chunk 69.
VAR_MAX_SCORE, used in chunk 69.
VAR_SCORE, used in chunk 69.

206c ⟨*Internal error string* 206c⟩≡ (207)

```
sInternalError:
    DC       "ZORK INTERNAL ERROR!"
```

Defines:
sInternalError, never used.

207 ⟨*routines* 207⟩≡ (202a)

```
        ORG     $0800
```

⟨*main* 26a⟩

⟨*Instruction tables* 109⟩

⟨*Do instruction* 112⟩
⟨*Execute instruction* 111⟩
⟨*Handle 0op instructions* 113b⟩
⟨*Handle 1op instructions* 114a⟩
⟨*Handle 2op instructions* 116a⟩
⟨*Get const byte* 120a⟩
⟨*Get const word* 120b⟩
⟨*Get var content in A* 122⟩
⟨*Store to var A* 124⟩
⟨*Get var content* 121⟩
⟨*Store and go to next instruction* 159a⟩
⟨*Store var* 123⟩
⟨*Handle branch* 161a⟩
⟨*Instruction rtrue* 184a⟩
⟨*Instruction rfalse* 183b⟩
⟨*Instruction print* 185a⟩
⟨*Printing a string literal* 68b⟩
⟨*Instruction print ret* 182b⟩
⟨*Instruction nop* 200a⟩
⟨*Instruction ret popped* 183a⟩
⟨*Instruction pop* 168b⟩
⟨*Instruction new line* 184b⟩
⟨*Instruction jz* 180b⟩
⟨*Instruction get sibling* 196⟩
⟨*Instruction get child* 188⟩
⟨*Instruction get parent* 190⟩
⟨*Instruction get prop len* 195⟩
⟨*Instruction inc* 169c⟩
⟨*Instruction dec* 170a⟩
⟨*Increment variable* 160a⟩
⟨*Decrement variable* 160b⟩
⟨*Instruction print addr* 185b⟩
⟨*Instruction illegal opcode* 158⟩
⟨*Instruction remove obj* 199a⟩
⟨*Remove object* 134a⟩
⟨*Instruction print obj* 186b⟩
⟨*Print object in A* 137⟩
⟨*Instruction ret* 129⟩
⟨*Instruction jump* 182a⟩
⟨*Instruction print paddr* 186c⟩
⟨*Print zstring and go to next instruction* 159b⟩
⟨*Instruction load* 165a⟩
⟨*Instruction not* 175b⟩

⟨*Instruction jl* 180a⟩
⟨*Instruction jg* 179a⟩
⟨*Instruction dec chk* 176b⟩
⟨*Instruction inc chk* 177a⟩
⟨*Instruction jin* 179b⟩
⟨*Instruction test* 181a⟩
⟨*Instruction or* 176a⟩
⟨*Instruction and* 175a⟩
⟨*Instruction test attr* 181b⟩
⟨*Instruction set attr* 199b⟩
⟨*Instruction clear attr* 187⟩
⟨*Instruction store* 166b⟩
⟨*Instruction insert obj* 197⟩
⟨*Instruction loadw* 165b⟩
⟨*Instruction loadb* 166a⟩
⟨*Instruction get prop* 191⟩
⟨*Instruction get prop addr* 194⟩
⟨*Instruction get next prop* 189⟩
⟨*Instruction add* 170b⟩
⟨*Instruction sub* 174c⟩
⟨*Instruction mul* 173⟩
⟨*Instruction div* 171⟩
⟨*Instruction mod* 172⟩
⟨*Instruction je* 177b⟩
⟨*Instruction call* 125⟩
⟨*Instruction storew* 167⟩
⟨*Instruction storeb* 168a⟩
⟨*Instruction put prop* 198⟩
⟨*Instruction sread* 73⟩
⟨*Skip separators* 79⟩
⟨*Separator checks* 80⟩
⟨*Get dictionary address* 90⟩
⟨*Match dictionary word* 91⟩
⟨*Instruction print char* 185c⟩
⟨*Instruction print num* 186a⟩
⟨*Print number* 103⟩
⟨*Print negative number* 104⟩
⟨*Instruction random* 174a⟩
⟨*Instruction push* 169b⟩
⟨*Instruction pull* 169a⟩
⟨*mulu16* 97⟩
⟨*divu16* 100⟩
⟨*Check sign* 96a⟩
⟨*Set sign* 96b⟩
⟨*negate* 95a⟩
⟨*Flip sign* 95b⟩
⟨*Get attribute pointer and mask* 138⟩
⟨*Get property pointer* 140⟩
⟨*Get property number* 141a⟩
⟨*Get property length* 141b⟩

⟨*Copy data to buff* 150a⟩
⟨*Instruction restore* 152b⟩
⟨*Copy data from buff* 153c⟩
⟨*Instruction quit* 201⟩
⟨*Internal error string* 206c⟩
⟨*brk* 31c⟩
⟨*Get random* 174b⟩

```
    HEX    00 00 00 00 00 00 00 00
    HEX    00 FC 19 00 00
```

Chapter 16

Defined Chunks

Chapter 17

Index