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, one section of the game originally written as 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 the revision 15 version of Zork I for the Apple II. The disk image used is 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

All files can be found on Github.

The source for this document, `main.nw`, is a literate programming document. This means the explanatory text is interspersed with source code. The assembly code and `LaTeX` file 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 code was reverse-engineered using Ghidra.

The assembly code was assembled using `dasm` using this command line:

```
dasm main.asm -Lmain.lst -omain.bin -f3 -v4
```

The document is written in LaTeX.

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.

1.3 Extracting the sections

The disk image contains the following sections. Note that the disk has 16 sectors per track, and we will refer to tracks and sectors only by `16 * track + sector`.

- Sector 0: `BOOT1`, target address `$0800`: The first stage boot loader.
- Sector 0-9: `BOOT2`, target address `$2200`: The second stage boot loader, loaded by `BOOT1`.
- Sector 16-41 (interleaved): `main`, target address `$0800`: The main program, loaded by `BOOT2`.
- Sector 48-119: Z-code for Zork I.

The sections can be extracted from the disk image using the following commands:

```
python -m extract --first 0 -n 1 -i "Zork I r15 (4am crack).dsk" -o boot1.bin
python -m extract --first 1 -n 9 -i "Zork I r15 (4am crack).dsk" -o boot2.bin
python -m extract --first 16 -n 26 -i "Zork I r15 (4am crack).dsk" -o main.bin --skew
```

We extract `BOOT2` only starting from sector 1, since the first page is just a copy of `BOOT1`.

Also note that the `--skew` option is used for `main.bin` because the sectors are interleaved on disk.

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 don'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.

12 ⟨*Macros* 12⟩≡ (206 207a) 13a ▷

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

Defines:
`STOW`, used in chunks 34–36, 59, 73, 102, 105, 108, 112, 113, 118b, 120b, 122c, 124, 130, 135b, 143a, 147, 149, 151–55, 157b, 158a, and 205.

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.

13a ⟨*Macros* 12⟩+≡ (206 207a) ◁12 13b▷

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

Defines:
STOW2, used in chunk 113.

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.

13b ⟨*Macros* 12⟩+≡ (206 207a) ◁13a 14a▷

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

Defines:
MOVB, used in chunks 24d, 42a, 49, 52b, 55a, 59, 72b, 75, 88a, 117, 131a, 133–35, and 185a.
STOB, used in chunks 23c, 28a, 34c, 35b, 39c, 41, 46, 47, 50, 52b, 53, 55b, 57, 59, 65, 66, 69, 72b, 73, 75, 77, 82, 85, 100b, 108, 111, 117, 119d, 122a, 125a, 131a, 133a, 136, 137, 152, and 163a.

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.

14a ⟨*Macros* 12⟩+≡ (206 207a) ◁13b 14b▷

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

Defines:
MOVW, used in chunks 36c, 46, 47, 50, 57, 60b, 65, 72b, 102, 105, 111, 117, 119d, 121, 131a, 133–36, 141b, 155b, 158b, 170b, 173b, 175–78, 180b, 181a, 183a, 184a, 186a, 187a, 189, 190, and 197.

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.

14b ⟨*Macros* 12⟩+≡ (206 207a) ◁14a 14c▷

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

Defines:
PSHW, used in chunks 57, 60b, 102, 105, 113, 128, 129, 139–41, 164a, and 201.

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`.

14c ⟨*Macros* 12⟩+≡ (206 207a) ◁14b 15a▷

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

Defines:
PULB, used in chunk 133b.

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.

15a ⟨*Macros* 12⟩+≡ (206 207a) ◁14c 15b▷

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

Defines:
PULW, used in chunks 57, 60b, 102, 105, 113, 128, 129, 140, 141a, 164a, and 201.

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.

15b ⟨*Macros* 12⟩+≡ (206 207a) ◁15a 16a▷

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

Defines:
INCW, used in chunks 44, 45, 112, 113, 141b, 142, 164a, 178a, and 198.

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`.

16a ⟨*Macros* 12⟩+≡ (206 207a) ◁15b 16b▷

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

Defines:
`ADDA`, used in chunks 95 and 135b.

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

16b ⟨*Macros* 12⟩+≡ (206 207a) ◁16a 17a▷

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

Defines:
`ADDAC`, used in chunk 198.

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.

17a ⟨*Macros* 12⟩+≡ (206 207a) ◁16b 17b▷

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

Defines:
ADDB, used in chunks 149 and 151b.

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

17b ⟨*Macros* 12⟩+≡ (206 207a) ◁17a 18a▷

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

Defines:
ADDB2, used in chunks 96 and 97.

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

For example, `ADDW $01FF, $0300, $0400` adds the 16-bit value at memory location `$01FF` to the 16-bit value at memory location `$0300`, and stores the result at memory location `$0400`.

18a ⟨*Macros* 12⟩+≡ (206 207a) ◁17b 18b▷

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

Defines:
ADDW, used in chunks 77, 94, 126, 128, 144, 169–72, and 174b.

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.

18b ⟨*Macros* 12⟩+≡ (206 207a) ◁18a 19a▷

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

Defines:
ADDWC, used in chunk 102.

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.

For example, SUBB $01FF, #$01 subtracts the immediate value #$01 from the 16-bit value at memory location $01FF and stores it back.

19a ⟨*Macros* 12⟩+≡ (206 207a) ◁18b 19b▷

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

Defines:
SUBB, used in chunks 43, 97, 135c, 164b, 167b, and 186a.

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

19b ⟨*Macros* 12⟩+≡ (206 207a) ◁19a 20a▷

```
    MACRO SUBB2
        SEC
        LDA     {1}
        SBC     {2}
        STA     {1}
        BCS     .continue
        DEC     {1}+1
.continue
    ENDM
```

Defines:
SUBB2, used in chunk 96b.

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.

For example, SUBW $01FF, $0300, $0400 subtracts the 16-bit value at memory location $0300 from the 16-bit value at memory location $01FF, and stores the result at memory location $0400.

20a ⟨*Macros* 12⟩+≡ (206 207a) ◁19b 20b▷

```
MACRO SUBW
    SEC
    LDA    {1}
    SBC    {2}
    STA    {3}
    LDA    {1}+1
    SBC    {2}+1
    STA    {3}+1
ENDM
```

Defines:
SUBW, used in chunks 98b, 178c, and 198.

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

For example, SUBWL #$01FF, $0300, $0400 subtracts the 16-bit value at memory location $0300 from the 16-bit value #$01FF, and stores the result at memory location $0400.

20b ⟨*Macros* 12⟩+≡ (206 207a) ◁20a 21a▷

```
MACRO SUBWL
    SEC
    LDA    <{1}
    SBC    {2}
    STA    {3}
    LDA    >{1}
    SBC    {2}+1
    STA    {3}+1
ENDM
```

Defines:
SUBWL, used in chunk 99.

2.6.7 ROLW, RORW

`ROLW` rotates a 16-bit memory location left.

21a ⟨*Macros* 12⟩+≡ (206 207a) ◁20b 21b▷

```
MACRO ROLW
    ROL    {1}
    ROL    {1}+1
ENDM
```

Defines:
`ROLW`, used in chunk 105.

`RORW` rotates a 16-bit memory location right.

21b ⟨*Macros* 12⟩+≡ (206 207a) ◁21a

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

Defines:
`RORW`, used in chunk 102.

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.

23a ⟨*BOOT1* 23a⟩≡ (206a) 23b ▷

```
    BYTE    #$01  ; Number of sectors in BOOT1. Almost always 1.
boot1:
    SUBROUTINE
```

Defines:
boot1, never used.

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

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.

23b ⟨*BOOT1* 23a⟩+≡ (206a) ◁ 23a 23c ▷

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

Uses IWMDATAPTR 208.

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.

23c ⟨*BOOT1* 23a⟩+≡ (206a) ◁ 23b 24a ▷

```
    LDA     IWMSLTNDX           ; The slot we're booting from, times 16.
    LSR
    LSR
    LSR
    LSR
    ORA     #$C0
    STA     RDSECT_PTR+1
    STOB    #$5C, RDSECT_PTR
```

Uses IWMSLTNDX 208, RDSECT_PTR 208, and STOB 13b.

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`.

24a ⟨*BOOT1* 23a⟩+≡ (206a) ◁23c 24b▷

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

Uses `BOOT1_SECTOR_NUM` 26b and `BOOT1_WRITE_ADDR` 26b.

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.

24b ⟨*BOOT1* 23a⟩+≡ (206a) ◁24a 24c▷

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

Defines:
`.already_initted`, never used.
Uses `BOOT1_SECTOR_NUM` 26b.

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.

24c ⟨*BOOT1* 23a⟩+≡ (206a) ◁24b 24d▷

```
    LDA     BOOT1_SECTOR_XLAT_TABLE,X
    STA     IWMSECTOR
```

Uses `BOOT1_SECTOR_XLAT_TABLE` 25b and `IWMSECTOR` 208.

Then we transfer the page of `BOOT1_WRITE_ADDR` into the page of `IWMDATAPTR`, decrement `BOOT1_SECTOR_NUM`, load up the X register with `IWMSLTNDX`, and do the read by jumping to the address in `RDSECT_PTR`. Remember that when that routine finishes, it jumps back to `boot1`.

24d ⟨*BOOT1* 23a⟩+≡ (206a) ◁24c 25a▷

```
    DEC     BOOT1_SECTOR_NUM
    MOVB    BOOT1_WRITE_ADDR+1, IWMDATAPTR+1
    DEC     BOOT1_WRITE_ADDR+1
    LDX     IWMSLTNDX
    JMP     (RDSECT_PTR)
```

Uses `BOOT1_SECTOR_NUM` 26b, `BOOT1_WRITE_ADDR` 26b, `IWMDATAPTR` 208, `IWMSLTNDX` 208, `MOVB` 13b, and `RDSECT_PTR` 208.

Once BOOT1 has finished loading, it jumps to what got loaded from sector 1. This is called BOOT2, the 2nd stage boot loader.

Note that because we read down to sector 0, and BOOT1_WRITE_ADDR got post-decremented, BOOT1_WRITE_ADDR points to one page before sector 0. Incrementing once would have it point to a copy of BOOT1, which we don't need. Therefore, we increment twice.

25a ⟨*BOOT1* 23a⟩+≡ (206a) ◁24d 25b▷

```
.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)
```

Defines:
  .go_to_boot2, never used.
Uses BOOT1_WRITE_ADDR 26b, INIT 208, IWMSLTNDX 208, SETKBD 208, and SETVID 208.

25b ⟨*BOOT1* 23a⟩+≡ (206a) ◁25a 26a▷

```
BOOT1_SECTOR_XLAT_TABLE:
    HEX     00 0D 0B 09 07 05 03 01
    HEX     0E 0C 0A 08 06 04 02 0F
```

Defines:
  BOOT1_SECTOR_XLAT_TABLE, used in chunk 24c.

The rest of the data in BOOT1 seems to contain unused garbage.

26a ⟨*BOOT1* 23a⟩+≡ (206a) ◁25b 26b▷

```
      HEX     00 20 64
      HEX     27 B0 08 A9 00 A8 8D 5D
      HEX     36 91 40 AD C5 35 4C D2
      HEX     26 AD 5D 36 F0 08 EE BD
      HEX     35 D0 03 EE BE 35 A9 00
      HEX     8D 5D 36 4C 46 25 8D BC
      HEX     35 20 A8 26 20 EA 22 4C
      HEX     7D 22 A0 13 B1 42 D0 14
      HEX     C8 C0 17 D0 F7 A0 19 B1
      HEX     42 99 A4 35 C8 C0 1D D0
      HEX     F6 4C BC 26 A2 FF 8E 5D
      HEX     36 D0 F6 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00
      HEX     20 58 FC A9 C2 20 ED FD  ; seems to be part of the monitor
      HEX     A9 01 20 DA FD A9 AD 20
      HEX     ED FD A9 00 20 DA FD 60
      HEX     00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00
```

26b ⟨*BOOT1* 23a⟩+≡ (206a) ◁26a

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

Defines:
BOOT1_SECTOR_NUM, used in chunk 24.
BOOT1_WRITE_ADDR, used in chunks 24 and 25a.

3.2 BOOT2

In normal DOS, BOOT2 is the 2nd stage boot loader. See Beneath Apple DOS, page 8-34, description of address $B700. However in this case, it looks like the programmers modified the first page of the standard BOOT2 loader so that it instead loads the main program from disk and then jumps to it.

Zork's BOOT2 loads 26 sectors starting from track 1 sector 0 into addresses $0800-$21FF, and then jumps to $0800. It also contains all the low-level disk routines from DOS, which includes RWTS, the read/write track/sector routine.

We will only look at the main part of BOOT2, not any of the low-level disk routines.

27 ⟨*BOOT2* 27⟩≡ (206b) 28a ▷

```
  boot2:
      SUBROUTINE

      LDA       #$1F
      STA       $7B

  .loop:
      LDA       #>boot2_iob            ; call RWTS with IOB
      LDY       #<boot2_iob
      JSR       RWTS_entry
      BCS       .loop                  ; on error, try again

      INC       sector_count
      LDA       sector_count
      CMP       #26
      BEQ       .start_main            ; done loading 26 sectors?

      INC       boot2_iob.buffer+1     ; increment page
      INC       boot2_iob.sector       ; increment sector and track
      LDA       boot2_iob.sector
      CMP       #16
      BNE       .loop

      LDA       #$00
      STA       boot2_iob.sector
      INC       boot2_iob.track
      JMP       .loop
```

Defines:
  boot2, never used.
Uses RWTS 209, RWTS_entry 236, boot2_iob 29a, and sector_count 28a.

28a ⟨*BOOT2* 27⟩+≡ (206b) ◁27 28b▷

```
.start_main:
    STOB    #$60, DEBUG_JUMP      ; an RTS instruction
    STOB    #16, SECTORS_PER_TRACK
    JSR     INIT
    JSR     SETVID
    JSR     SETKBD
    JMP     main

sector_count:
    HEX     00
```

Defines:
sector_count, used in chunk 27.
Uses DEBUG_JUMP 209, INIT 208, SECTORS_PER_TRACK 209, SETKBD 208, SETVID 208, STOB 13b, and main 33a.

A zeroed out area:

28b ⟨*BOOT2* 27⟩+≡ (206b) ◁28a 29a▷

```
BACK_TO_BOOT2:
    HEX     00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00
```

The RWTS parameter list (I/O block):

29a ⟨*BOOT2* 27⟩+≡ (206b) ◁28b 29b▷

```
boot2_iob:
    HEX     01          ; table type, must be 1
    HEX     60          ; slot times 16
    HEX     01          ; drive number
    HEX     00          ; volume number
boot2_iob.track:
    HEX     01          ; track number
boot2_iob.sector:
    HEX     00          ; sector number
boot2_iob.dct_addr:
    WORD    boot2_dct   ; address of device characteristics table
boot2_iob.buffer:
    WORD    #$0800      ; address of buffer
    HEX     00 00
boot2_iob.command:
    HEX     01          ; command byte (read)
    HEX     00          ; return code
    HEX     00          ; last volume number
    HEX     60          ; last slot times 16
    HEX     01          ; last drive number
```

Defines:
- boot2_iob, used in chunk 27.
- boot2_iob.buffer, never used.
- boot2_iob.command, never used.
- boot2_iob.dct_addr, never used.
- boot2_iob.sector, never used.
- boot2_iob.track, never used.

Uses boot2_dct 29b.

The Device Characteristics Table:

29b ⟨*BOOT2* 27⟩+≡ (206b) ◁29a 30a▷

```
boot2_dct:
    HEX     00         ; device type, must be 0
    HEX     01         ; phases per track, must be 1
    WORD    #$D8EF     ; motor on time count
```

Defines:
- boot2_dct, used in chunk 29a.

Some bytes apparently left over and unzeroed, and then zeros to the end of the page.

30a ⟨*BOOT2* 27⟩+≡ (206b) ◁29b 30b▷

```
HEX     00 00 00
HEX     00 00 00 00 00 00 DE 00
HEX     00 00 02 00 01 01 00 00
HEX     00 00 00 00 00 00 00 00
HEX     00 00 00 00 00 00 00 00
HEX     00 00 00 00 00 00 00 00
```

30b ⟨*BOOT2* 27⟩+≡ (206b) ◁30a

⟨*RWTS routines* 250⟩

Chapter 4

The Z-image

The Z-machine operates on the Z-image of the game, and executes Z-code in Z-memory. The Z-image on disk is divided into three consecutive parts: dynamic, static, and high.

Dynamic memory is memory that may be changed by the game, and is part of the game state when saving and restoring. It consists of the Z-image header (the first page of the Z-image) and any other modifiable data such as the object hierarchy and global variables.

Static memory is memory that is not changed by the game, so it does not have to be saved as part of a saved game. It consists of things like strings, and the game dictionary (list of words accepted by the parser).

High memory is memory that is not loaded into memory when the game starts, but is instead read from disk on demand. This is where the Z-code lives.

When starting the game, the Z-image dynamic and static parts are loaded into 6502 memory. The high part is not loaded, but is instead read from disk on demand as Z-code is executed.

For this version of Zork I, these are the Z-addresses (addresses relative to the beginning of the Z-header) for various parts of the Z-image. These addresses are stored in the Z-header in big-endian order, and are used by the Z-machine to locate the various parts of the Z-image:

- Static memory base: `$2CDC`
- High memory base: `$476C`
- Starting Z program counter: `$4859`

- Start of abbreviations table: `$00CA`
- Start of object table: `$010A`
- Start of global variables: `$20DE`
- Start of dictionary: `$3686`

Chapter 5

The main program

This is the Z-machine proper.

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

33a ⟨*main* 33a⟩≡ (213) 33b ▷

```
main:
    SUBROUTINE

    CLD
    LDA     #$00
    LDX     #$80

.clear:
    STA     $00,X
    INX
    BNE     .clear
```

Defines:
`main`, used in chunks 28a, 36c, 38a, 47, 50, 204b, and 206b.

And we reset the 6502 stack pointer.

33b ⟨*main* 33a⟩+≡ (213) ◁ 33a 34a ▷

```
    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.

34a ⟨*main* 33a⟩+≡ (213) ◁33b 34b▷

```
  .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
```

Uses PRINTER_CSW 209.

Next, we set ZCODE_PAGE_VALID to zero. This means that any access to Z-code will result in its page needing to be located, either in the cache, or loaded from disk.

34b ⟨*main* 33a⟩+≡ (213) ◁34a 34c▷

```
      LDA       #$00
      STA       ZCODE_PAGE_VALID
      STA       ZCODE_PAGE_VALID2
```

Uses ZCODE_PAGE_VALID 209 and ZCODE_PAGE_VALID2 209.

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

34c ⟨*main* 33a⟩+≡ (213) ◁34b 34d▷

```
      STOB      #$01, STACK_COUNT
      STOW      #$03E8, Z_SP
      STOB      #$FF, ZCHAR_SCRATCH1+6
```

Uses STACK_COUNT 209, STOB 13b, STOW 12, ZCHAR_SCRATCH1 209, and Z_SP 209.

There are two page cache tables, PAGE_L_TABLE and PAGE_H_TABLE, which are set to $2200 and $2280, respectively. These are used to map Z-code 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.

34d ⟨*main* 33a⟩+≡ (213) ◁34c 35a▷

```
      STOW      #$2200, PAGE_L_TABLE
      STOW      #$2280, PAGE_H_TABLE
      STOW      #$2300, NEXT_PAGE_TABLE
      STOW      #$2380, PREV_PAGE_TABLE
```

Uses NEXT_PAGE_TABLE 209, PAGE_H_TABLE 209, PAGE_L_TABLE 209, PREV_PAGE_TABLE 209, and STOW 12.

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.

The linked list is a most-recently-used cache. When a page of Z-code is accessed, it is placed on the front of the list.

35a ⟨*main* 33a⟩+≡ (213) ◁34d 35b▷

```
    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 209, PAGE_H_TABLE 209, PAGE_L_TABLE 209, and PREV_PAGE_TABLE 209.

To complete initialization of the linked list, we set FIRST_Z_PAGE to 0 (the head of the list), LAST_Z_PAGE to #$7F (the tail of the list).

35b ⟨*main* 33a⟩+≡ (213) ◁35a 35c▷

```
    STOB    #$00, FIRST_Z_PAGE
    STOB    #$7F, LAST_Z_PAGE
```

Uses FIRST_Z_PAGE 209, LAST_Z_PAGE 209, and STOB 13b.

Now we set Z_HEADER_ADDR to $2C00. Z_HEADER_ADDR is the address in memory where the Z-image starts, and where the Z-header is located.

35c ⟨*main* 33a⟩+≡ (213) ◁35b 36a▷

```
    STOW    #$2C00, Z_HEADER_ADDR
```

Uses STOW 12.

Then we clear the screen.

36a ⟨*main* 33a⟩+≡ (213) ◁35c 36c▷
```
    JSR     do_reset_window
```
Uses `do_reset_window` 36b.

36b ⟨*Do reset window* 36b⟩≡ (212c)
```
  do_reset_window:
    JSR     reset_window
    RTS
```
Defines:
`do_reset_window`, used in chunk 36a.
Uses `reset_window` 55b.

Next, we read the first page of the Z-image from disk into memory. This is the Z-header, which 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`. This means that the Z-image is always stored from sector 48 (`=3*16`) on.

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.

36c ⟨*main* 33a⟩+≡ (213) ◁36a 37▷
```
  .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` 14a, `SCRATCH1` 209, `SCRATCH2` 209, `STOW` 12, `main` 33a, and `read_from_sector` 112.

If there was no error reading the image header, we write `#$FF` into the low byte of the base of high memory in the header, and then we load the page (high byte) of the base of high memory from the header and store it (plus 1) in `NUM_IMAGE_PAGES`. That is, `NUM_IMAGE_PAGES` is the number of pages in the dynamic and static parts of the Z-image.

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

37 ⟨*main* 33a⟩+≡ (213) ◁36c 38a▷

```
  .no_error:
      LDY      #HEADER_HIMEM_BASE+1
      LDA      #$FF
      STA      (Z_HEADER_ADDR),Y      ; low byte of high memory base always FF.
      DEY
      LDA      (Z_HEADER_ADDR),Y      ; page
      STA      NUM_IMAGE_PAGES
      INC      NUM_IMAGE_PAGES
```

Uses `NUM_IMAGE_PAGES` 209.

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

38a ⟨*main* 33a⟩+≡ (213) ◁37 38b▷
```
    LDA     #$00                ; sector = 0

.read_another_sector:
    CLC                         ; ++sector
    ADC     #$01
    TAX
    ADC     Z_HEADER_ADDR+1     ; dest_addr = Z_HEADER_ADDR + 256*sector
    STA     SCRATCH2+1
    LDA     Z_HEADER_ADDR
    STA     SCRATCH2
    TXA
    CMP     NUM_IMAGE_PAGES
    BEQ     .check_debug_flag   ; done loading?
    PHA                         ; read_sector = sector
    STA     SCRATCH1
    LDA     #$00
    STA     SCRATCH1+1
    JSR     read_from_sector    ; read_from_sector(read_sector, dest_addr)

    ; 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 209, SCRATCH1 209, SCRATCH2 209, main 33a, and read_from_sector 112.

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.

38b ⟨*main* 33a⟩+≡ (213) ◁38a 39c▷
```
.check_debug_flag:
    LDY     #HEADER_FLAGS1
    LDA     (Z_HEADER_ADDR),Y
    AND     #$01
    EOR     #$01
    BEQ     .brk
```
Uses brk 39b.

39a ⟨*die* 39a⟩≡ (42b)

```
.brk:
    JSR     brk
```

Uses brk 39b.

39b ⟨*brk* 39b⟩≡ (213)

```
brk:
    BRK
```

Defines:
brk, used in chunks 38b, 39a, 41, 43, 44, 162, 181b, 197, and 202.

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, big-endian. For Zork I, Z_PC becomes #$004859.

39c ⟨*main* 33a⟩+≡ (213) ◁38b 40▷

```
.store_initial_z_pc:
    LDY     #HEADER_INITIAL_ZPC+1
    LDA     (Z_HEADER_ADDR),Y
    STA     Z_PC
    DEY
    LDA     (Z_HEADER_ADDR),Y
    STA     Z_PC+1
    STOB    #$00, Z_PC+2
```

Uses STOB 13b and Z_PC 209.

Next, we load GLOBAL_ZVARS_ADDR and Z_ABBREV_TABLE from the header addresses. Again, these are big-endian 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.

40 ⟨*main* 33a⟩+≡ (213) ◁39c 41▷

```
  .store_z_global_vars_addr:
      LDY       #HEADER_GLOBALVARS_ADDR+1
      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       #HEADER_ABBREVS_ADDR+1
      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 209 and Z_ABBREV_TABLE 209.

Next, we set HIGH_MEM_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, HIGH_MEM_ADDR is $7400.

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

Recall earlier that we set up a linked list of cached Z-code pages. This list was 128 pages. However, it is not true that the physical memory can store that many pages. Once we find the last viable RAM page, we determine the number of pages that can be stored in memory.

Thus, we store the difference between the last viable RAM page and the page of HIGH_MEM_ADDR into LAST_Z_PAGE, and the same, plus 1, in NUM_PAGE_TABLE_ENTRIES. We also set the next page table entry of the last page to #$FF.

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

41 ⟨*main* 33a⟩+≡ (213) ◁40 42b▷

```
    STOB    #$00, HIGH_MEM_ADDR
    LDA     NUM_IMAGE_PAGES
    CLC
    ADC     Z_HEADER_ADDR+1
    STA     HIGH_MEM_ADDR+1
    JSR     locate_last_ram_page
    SEC
    SBC     HIGH_MEM_ADDR+1
    BCC     .brk
    TAY
    INY
    STY     NUM_PAGE_TABLE_ENTRIES
    TAY
    STY     LAST_Z_PAGE
    LDA     #$FF
    STA     (NEXT_PAGE_TABLE),Y
```

Uses HIGH_MEM_ADDR 209, LAST_Z_PAGE 209, NEXT_PAGE_TABLE 209, NUM_IMAGE_PAGES 209, STOB 13b, and brk 39b.

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`.

42a ⟨*Locate last RAM page* 42a⟩≡ (213)

```
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` 13b and `SCRATCH2` 209.

The final step of the main routine is to start the interpreter loop by executing the first instruction in z-code.

42b ⟨*main* 33a⟩+≡ (213) ◁41

```
    JMP       do_instruction

⟨die 39a⟩
```

Uses `do_instruction` 117.

Chapter 6

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 values on the stack becomes `#$B4` (180), the program will hit a BRK instruction.

43 ⟨*Push* 43⟩≡ (213)

```
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 128, 129, 131–33, and 173b.
Uses SCRATCH2 209, STACK_COUNT 209, SUBB 19a, Z_SP 209, and brk 39b.

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.

44 ⟨*Pop* 44⟩≡ (213)

```
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 126, 129, 135, 172b, 173a, and 187a.
Uses INCW 15b, SCRATCH2 209, STACK_COUNT 209, Z_SP 209, and brk 39b.

Chapter 7

Z-code and the page cache

As mentioned earlier, ZCODE_PAGE_VALID is a flag that indicates whether the location in 6502 memory of the current page of Z-code is known. If it is known, it will be in ZCODE_PAGE_ADDR. Otherwise it has to be found, either in the page cache or on disk.

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 plus the low byte of Z_PC and then incrementing Z_PC.

Of course, if the low byte of Z_PC ends up as 0 after the increment, it means we will be crossing over into another page, which means we must invalidate the code page.

When the Z-machine starts, ZCODE_PAGE_VALID is zero.

45 ⟨*Get next code byte* 45⟩≡ (213) 46 ▷

```
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  ; will next byte be in next page?
    RTS

.invalidate_zcode_page:
    LDY     #$00
    STY     ZCODE_PAGE_VALID
```

```
    INCW    Z_PC+1
    RTS
```

Defines:
`get_next_code_byte`, used in chunks 46, 117, 118a, 125, 126, 128, 131c, 132, 165, and 166.
Uses `INCW` 15b, `ZCODE_PAGE_ADDR` 209, `ZCODE_PAGE_VALID` 209, and `Z_PC` 209.

Z_PC is a 24-bit address relative to the beginning of the Z-header. This means that Z-code could in theory be stored within the dynamic or static areas of the Z-image. If it is, then we can easily access it, since it was loaded when the game was initially loaded.

In either case, the page we just accessed is placed in the front of the linked list of cached pages. This means that the list is a most-recently-used cache.

46 ⟨*Get next code byte* 45⟩+≡ (213) ◁45 47▷

```
  .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     ; Z_PC is in dynamic or static memory

  .find_pc_page_in_page_table:
    MOVW    Z_PC+1, SCRATCH2
    JSR     find_index_of_page_table
    STA     PAGE_TABLE_INDEX
    BCS     .not_found_in_page_table   ; not loaded from disk yet

  .set_page_first:
    JSR     set_page_first     ; move page to head of list
    CLC
    LDA     PAGE_TABLE_INDEX
    ADC     NUM_IMAGE_PAGES

  .set_page_addr:
    CLC
    ADC     Z_HEADER_ADDR+1
    STA     ZCODE_PAGE_ADDR+1
    STOB    #$00, ZCODE_PAGE_ADDR
    STOB    #$FF, ZCODE_PAGE_VALID ; code page is now valid
    JMP     get_next_code_byte
```

Defines:
`.zcode_page_invalid`, never used.
Uses `MOVW` 14a, `NUM_IMAGE_PAGES` 209, `PAGE_TABLE_INDEX` 209, `SCRATCH2` 209, `STOB` 13b, `ZCODE_PAGE_ADDR` 209, `ZCODE_PAGE_VALID` 209, `Z_PC` 209, `find_index_of_page_table` 48, `get_next_code_byte` 45, and `set_page_first` 49.

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

47 ⟨*Get next code byte* 45⟩+≡ (213) ◁46

```
.not_found_in_page_table:
    CMP     PAGE_TABLE_INDEX2
    BNE     .read_from_disk
    STOB    #$00, ZCODE_PAGE_VALID2

.read_from_disk:
    MOVW    HIGH_MEM_ADDR, SCRATCH2
    LDA     PAGE_TABLE_INDEX
    CLC
    ADC     SCRATCH2+1
    STA     SCRATCH2+1
    MOVW    Z_PC+1, SCRATCH1
    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 HIGH_MEM_ADDR 209, MOVW 14a, PAGE_H_TABLE 209, PAGE_L_TABLE 209, PAGE_TABLE_INDEX 209, PAGE_TABLE_INDEX2 209, SCRATCH1 209, SCRATCH2 209, STOB 13b, ZCODE_PAGE_VALID2 209, Z_PC 209, good_read 228, main 33a, read_from_sector 112, and set_page_first 49.

Given a page-aligned address in SCRATCH2, the find_index_of_page_table 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.

48 ⟨*Find index of page table* 48⟩≡ (213)

```
find_index_of_page_table:
    SUBROUTINE

    LDX      NUM_PAGE_TABLE_ENTRIES
    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 46 and 50.
Uses LAST_Z_PAGE 209, PAGE_H_TABLE 209, PAGE_L_TABLE 209, and SCRATCH2 209.

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.

49 ⟨*Set page first* 49⟩≡ (213)

```
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:
    MOVB    SCRATCH2+1, LAST_Z_PAGE    ; LAST_Z_PAGE = SCRATCH2H
    RTS
```

Defines:
`set_page_first`, used in chunks 46, 47, and 50.

Uses `FIRST_Z_PAGE` 209, `LAST_Z_PAGE` 209, `MOVB` 13b, `NEXT_PAGE_TABLE` 209, `PREV_PAGE_TABLE` 209, and `SCRATCH2` 209.

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're forced to separate out the bytes into Z_PC2_HH, Z_PC2_H, and Z_PC2_L.

50 ⟨*Get next code byte 2* 50⟩≡ (213)

```
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:
    MOVW    Z_PC2_H, SCRATCH2
    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
    STOB     #$00, ZCODE_PAGE_ADDR2
    STOB     #$FF, ZCODE_PAGE_VALID2
    JMP      get_next_code_byte2

.not_found_in_page_table:
    CMP      PAGE_TABLE_INDEX
    BNE      .read_from_disk
    STOB     #$00, ZCODE_PAGE_VALID

.read_from_disk:
    MOVW     HIGH_MEM_ADDR, SCRATCH2
    LDA      PAGE_TABLE_INDEX2
    CLC
    ADC      SCRATCH2+1
    STA      SCRATCH2+1
    MOVW     Z_PC2_H, SCRATCH1
    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 52a and 170a.

Uses HIGH_MEM_ADDR 209, MOVW 14a, NUM_IMAGE_PAGES 209, PAGE_H_TABLE 209, PAGE_L_TABLE 209, PAGE_TABLE_INDEX 209, PAGE_TABLE_INDEX2 209, SCRATCH1 209, SCRATCH2 209, STOB 13b, ZCODE_PAGE_ADDR2 209, ZCODE_PAGE_VALID 209, ZCODE_PAGE_VALID2 209, Z_PC2_H 209, Z_PC2_HH 209, Z_PC2_L 209, find_index_of_page_table 48, good_read 228, main 33a, read_from_sector 112, and set_page_first 49.

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.

52a ⟨*Get next code word* 52a⟩≡ (213)
```
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 65 and 169b.
Uses SCRATCH2 209 and get_next_code_byte2 50.

The load_address routine copies SCRATCH2 to Z_PC2.

52b ⟨*Load address* 52b⟩≡ (213)
```
load_address:
    SUBROUTINE

    MOVB       SCRATCH2, Z_PC2_L
    MOVB       SCRATCH2+1, Z_PC2_H
    STOB       #$00, Z_PC2_HH
```
Defines:
load_address, used in chunks 141b, 169b, 170a, and 189b.
Uses MOVB 13b, SCRATCH2 209, STOB 13b, Z_PC2_H 209, Z_PC2_HH 209, and Z_PC2_L 209.

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

53 ⟨*Load packed address* 53⟩≡ (213)

```
invalidate_zcode_page2:
    SUBROUTINE

    STOB     #$00, 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 69 and 190c.

Uses `SCRATCH2` 209, `STOB` 13b, `ZCODE_PAGE_VALID2` 209, `Z_PC2_H` 209, `Z_PC2_HH` 209, and `Z_PC2_L` 209.

Chapter 8

I/O

8.1 Strings and output

8.1.1 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.

54 ⟨*Output string to console* 54⟩≡ (213)

```
cout_string:
    SUBROUTINE

    LDY     #$00

.loop:
```

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

Defines:
cout_string, used in chunks 59, 73, and 149.
Uses COUT1 208 and SCRATCH2 209.

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.

55a ⟨*Home* 55a⟩≡ (213)

```
home:
    SUBROUTINE

    JSR     HOME
    MOVB    WNDTOP, CURR_LINE
    RTS
```

Defines:
home, used in chunks 55b and 147.
Uses CURR_LINE 209, HOME 208, MOVB 13b, and WNDTOP 208.

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.

55b ⟨*Reset window* 55b⟩≡ (213)

```
reset_window:
    SUBROUTINE

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

Defines:
reset_window, used in chunk 36b.
Uses INVFLG 208, PROMPT 208, STOB 13b, WNDBTM 208, WNDLFT 208, WNDTOP 208, WNDWDTH 208, and home 55a.

8.1.2 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`.

56 ⟨*Dump buffer to screen* 56⟩≡ (213)

```
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 58 and 73.

Uses `BUFF_AREA` 209, `BUFF_END` 209, and `COUT1` 208.

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.

57 ⟨*Dump buffer to printer* 57⟩≡ (213)

```
printer_card_initialized_flag:
    BYTE     00

dump_buffer_to_printer:
    SUBROUTINE

    PSHW     CSW
    MOVW     PRINTER_CSW, CSW
    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
    STOB     #$91, $0779      ; 'Q' into scratchpad RAM for slot 1.
    LDA      #$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:
    MOVW     CSW, PRINTER_CSW
    PULW     CSW
    RTS
```

Defines:
`dump_buffer_to_printer`, used in chunks 58 and 75.
`printer_card_initialized_flag`, never used.

Uses `BUFF_AREA` 209, `BUFF_END` 209, `COUT` 208, `CSW` 208, `MOVW` 14a, `PRINTER_CSW` 209, `PSHW` 14b, `PULW` 15a, and `STOB` 13b.

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`.

58 ⟨*Dump buffer line* 58⟩≡ (213)

```
dump_buffer_line:
    SUBROUTINE

    LDY      #HEADER_FLAGS2+1
    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 60a, 73, 75, 149, 151a, and 152.

Uses `HEADER_FLAGS2` 211a, `dump_buffer_to_printer` 57, and `dump_buffer_to_screen` 56.

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.

59 ⟨*Dump buffer with more* 59⟩≡ (213) 60a ▷

```
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

    STOB    #$3F, INVFLG
    JSR     cout_string    ; print [MORE] in inverse text
    STOB    #$FF, INVFLG

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

.good_to_go:
```

Defines:
dump_buffer_with_more, used in chunks 61, 62b, 147, 149, 151, 152, 204b, and 205.

Uses CH 208, CLREOL 208, CURR_LINE 209, INVFLG 208, MOVB 13b, RDKEY 208, SCRATCH2 209, STOB 13b, STOW 12, WNDBTM 208, WNDTOP 208, and cout_string 54.

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.

60a ⟨*Dump buffer with more* 59⟩+≡ (213) ◁59 60b▷

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

    .skip_newline:
```

Uses `BUFF_END` 209, `COUT1` 208, `WNDWDTH` 208, and `dump_buffer_line` 58.

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.

60b ⟨*Dump buffer with more* 59⟩+≡ (213) ◁60a 60c▷

```
        LDY     #HEADER_FLAGS2+1
        LDA     (Z_HEADER_ADDR),Y
        AND     #$01
        BEQ     .reset_buffer_end

        PSHW    CSW
        MOVW    PRINTER_CSW, CSW

        LDA     #$8D
        JSR     COUT

        MOVW    CSW, PRINTER_CSW
        PULW    CSW

    .reset_buffer_end:
```

Uses `COUT` 208, `CSW` 208, `HEADER_FLAGS2` 211a, `MOVW` 14a, `PRINTER_CSW` 209, `PSHW` 14b, and `PULW` 15a.

The last step is to set `BUFF_END` to zero.

60c ⟨*Dump buffer with more* 59⟩+≡ (213) ◁60b

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

Uses `buffer_char_set_buffer_end` 61.

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.

61 ⟨*Buffer a character* 61⟩≡ (213) 62a ▷

```
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 63b, 70a, 71c, 73, 108, 109, 148a, 150, 186b, 188b, and 189c.
`buffer_char_set_buffer_end`, used in chunk 60c.

Uses `BUFF_AREA` 209, `BUFF_END` 209, `WNDWDTH` 208, and `dump_buffer_with_more` 59.

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.

62a ⟨*Buffer a character* 61⟩+≡ (213) ◁61 62b▷

```
      LDA      #$A0  ; normal space

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

  .endloop:
```

Uses BUFF_AREA 209 and WNDWDTH 208.

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.

62b ⟨*Buffer a character* 61⟩+≡ (213) ◁62a 63a▷

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

Uses BUFF_END 209, BUFF_LINE_LEN 209, and dump_buffer_with_more 59.

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.

63a ⟨*Buffer a character* 61⟩+≡ (213) ◁62b

```
.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 209, BUFF_END 209, BUFF_LINE_LEN 209, and WNDWDTH 208.

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.

63b ⟨*Print ASCII string* 63b⟩≡ (212c)

```
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 147, 149, 151a, 152, and 205.
Uses SCRATCH2 209, SCRATCH3 209, and buffer_char 61.

8.1.3 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.

64 ⟨*Get alphabet* 64⟩≡ (213)

```
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 67a and 68.
Uses LOCKED_ALPHABET 209 and SHIFT_ALPHABET 209.

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.

65 ⟨*Get next zchar* 65⟩≡ (213)

```
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
    MOVW    SCRATCH2, ZCHARS_L
    LDA     ZCHARS_H
    LSR
    LSR
    AND     #$1F
    CLC
    RTS

.check_for_char_2:
    SEC
    SBC     #$01
    BNE     .check_for_last
    STOB    #$02, 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:
    STOB    #$00, ZDECOMPRESS_STATE
    LDA     ZCHARS_H
    BPL     .get_char_3
    STOB    #$FF, ZDECOMPRESS_STATE

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

Defines:
get_next_zchar, used in chunks 67a, 69, and 72a.
Uses MOVW 14a, SCRATCH2 209, STOB 13b, ZCHARS_H 209, ZCHARS_L 209, ZDECOMPRESS_STATE 209, and get_next_code_word 52a.

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.

66 ⟨*Print zstring* 66⟩≡ (213) 67a ▷

```
print_zstring:
    SUBROUTINE

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

Defines:
print_zstring, used in chunks 69, 72b, 141b, and 163b.
Uses LOCKED_ALPHABET 209, SHIFT_ALPHABET 209, STOB 13b, and ZDECOMPRESS_STATE 209.

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.

67a ⟨*Print zstring* 66⟩+≡ (213) ◁66

```
.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 70a⟩
```

Uses `SCRATCH3` 209, `get_alphabet` 64, and `get_next_zchar` 65.

67b ⟨*Printing a space* 67b⟩≡ (213)

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

Defines:
`.space`, never used.

68 ⟨*Shifting alphabets 68*⟩≡ (213)

```
  .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` 107, `LOCKED_ALPHABET` 209, `SCRATCH3` 209, `SHIFT_ALPHABET` 209,
  and `get_alphabet` 64.

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.

69 ⟨*Printing an abbreviation* 69⟩≡ (213)

```
  .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
        STOB    #$FF, SHIFT_ALPHABET ; Resets any temporary shift
        JMP     .loop
```

Defines:
`.abbreviation`, never used.

Uses `LOCKED_ALPHABET` 209, `SCRATCH2` 209, `SHIFT_ALPHABET` 209, `STOB` 13b, `ZCHARS_H` 209, `ZCHARS_L` 209, `ZCODE_PAGE_VALID2` 209, `ZDECOMPRESS_STATE` 209, `Z_ABBREV_TABLE` 209, `Z_PC2_H` 209, `Z_PC2_HH` 209, `Z_PC2_L` 209, `get_next_zchar` 65, `load_packed_address` 53, and `print_zstring` 66.

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`.

70a ⟨*Print the zchar* 70a⟩≡ (67a) 70b ▷

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

    .add_ascii_offset:
        CLC
        ADC     SCRATCH3

    .printchar:
        JSR     buffer_char
        JMP     .loop
```

Uses `SCRATCH3` 209 and `buffer_char` 61.

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

70b ⟨*Print the zchar* 70a⟩+≡ (67a) ◁ 70a 71b ▷

```
    .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.

71a ⟨*A2 table* 71a⟩≡ (213)

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

Defines:
`a2_table`, used in chunks 71b and 92b.

71b ⟨*Print the zchar* 70a⟩+≡ (67a) ◁70b

```
.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` 209 and `a2_table` 71a.

71c ⟨*Printing a CRLF* 71c⟩≡ (213)

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

Defines:
`.crlf`, never used.
Uses `buffer_char` 61.

72a ⟨*Printing a 10-bit ZSCII character* 72a⟩≡ (213)

```
.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` 209 and `get_next_zchar` 65.

`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`.

72b ⟨*Printing a string literal* 72b⟩≡ (213)

```
print_string_literal:
    SUBROUTINE

    MOVB     Z_PC, Z_PC2_L
    MOVB     Z_PC+1, Z_PC2_H
    MOVB     Z_PC+2, Z_PC2_HH
    STOB     #$00, ZCODE_PAGE_VALID2
    JSR      print_zstring
    MOVB     Z_PC2_L, Z_PC
    MOVB     Z_PC2_H, Z_PC+1
    MOVB     Z_PC2_HH, Z_PC+2
    MOVB     ZCODE_PAGE_VALID2, ZCODE_PAGE_VALID
    MOVW     ZCODE_PAGE_ADDR2, ZCODE_PAGE_ADDR
    RTS
```

Uses `MOVB` 13b, `MOVW` 14a, `STOB` 13b, `ZCODE_PAGE_ADDR` 209, `ZCODE_PAGE_ADDR2` 209, `ZCODE_PAGE_VALID` 209, `ZCODE_PAGE_VALID2` 209, `Z_PC` 209, `Z_PC2_H` 209, `Z_PC2_HH` 209, `Z_PC2_L` 209, and `print_zstring` 66.

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.

73 ⟨*Print status line* 73⟩≡ (213)

```
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
    STOB      #$3F, INVFLG
    JSR       CLREOL

    LDA       #VAR_CURR_ROOM
    JSR       var_get
    JSR       print_obj_in_A
    JSR       dump_buffer_to_screen

    STOB      #25, CH
    STOW      #sScore, SCRATCH2
    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

    STOB      #$FF, INVFLG
```

```
    PLA
    STA     CV
    PLA
    STA     CH
    JSR     VTAB
    RTS
```

Defines:
  print_status_line, used in chunk 77.
  sScore, never used.
Uses CH 208, CLREOL 208, CV 208, INVFLG 208, SCRATCH2 209, STOB 13b, STOW 12, VAR_CURR_ROOM 211b, VAR_MAX_SCORE 211b, VAR_SCORE 211b, VTAB 208, buffer_char 61, cout_string 54, dump_buffer_line 58, dump_buffer_to_screen 56, print_number 108, print_obj_in_A 141b, and var_get 127.

8.1.4 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.

75 ⟨*Read line* 75⟩≡ (213)

```
read_line:
    SUBROUTINE

    JSR     dump_buffer_line
    MOVB    WNDTOP, 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+1
    LDA     (Z_HEADER_ADDR),Y
    AND     #$01                ; Mask for transcript on
    BEQ     .continue
    TXA
    STA     BUFF_END
    JSR     dump_buffer_to_printer
    STOB    #$00, 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 77.

Uses `BUFF_AREA` 209, `BUFF_END` 209, `CURR_LINE` 209, `GETLN1` 208, `HEADER_FLAGS2` 211a, `MOVB` 13b, `OPERAND0` 209, `STOB` 13b, `WNDTOP` 208, `dump_buffer_line` 58, and `dump_buffer_to_printer` 57.

8.1.5 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.

77 ⟨*Instruction sread* 77⟩≡ (213) 78a▷

```
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
    STOB    #$00, SCRATCH3 ; SCRATCH3L = 0  (char count)
    LDY     #$01
    LDA     #$00            ; store 0 in the parse buffer + 1.
    STA     (OPERAND1),Y
    STOB    #$02, TOKEN_IDX
    STOB    #$01, INPUT_PTR
```

Defines:
`instr_sread`, used in chunk 114.

Uses `ADDW` 18a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH3` 209, `STOB` 13b, `print_status_line` 73, and `read_line` 75.

Loop:

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

78a ⟨*Instruction sread* 77⟩+≡ (213) ◁77 78b▷

```
.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 209 and do_instruction 117.

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

78b ⟨*Instruction sread* 77⟩+≡ (213) ◁78a 78c▷

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

Uses SCRATCH3 209 and do_instruction 117.

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

78c ⟨*Instruction sread* 77⟩+≡ (213) ◁78b 79a▷

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

Uses SCRATCH3 209 and skip_separators 83.

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

79a ⟨*Instruction sread* 77⟩+≡ (213) ◁78c 79b▷

```
  .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 209 and ZCHAR_SCRATCH1 209.

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.

79b ⟨*Instruction sread* 77⟩+≡ (213) ◁79a 80a▷

```
      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 209, OPERAND1 209, SCRATCH3 209, is_dict_separator 84, and is_std_separator 84.

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.

80a ⟨*Instruction sread* 77⟩+≡ (213) ◁79b 80b▷

```
  .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` 209, `SCRATCH3` 209, `ZCHAR_SCRATCH1` 209, and `is_separator` 84.

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.

80b ⟨*Instruction sread* 77⟩+≡ (213) ◁80a 81▷

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

Uses `SCRATCH3` 209, `ZCHAR_SCRATCH1` 209, and `is_dict_separator` 84.

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.

81 ⟨*Instruction sread* 77⟩+≡ (213) ◁80b 82▷

```
.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 209 and SCRATCH3 209.

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.

82 ⟨*Instruction sread* 77⟩+≡ (213) ◁81

```
        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
        STOB    #$00, SCRATCH3
        JMP     .loop_word
```

Uses `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH3` 209, `STOB` 13b, `ascii_to_zchar` 85, and `match_dictionary_word` 95.

Separators

83 ⟨*Skip separators 83*⟩≡ (213)

```
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 78c.

Uses `OPERAND0` 209, `SCRATCH3` 209, and `is_separator` 84.

84 ⟨*Separator checks 84*⟩≡ (213)

```
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 79b and 80b.
is_separator, used in chunks 80a and 83.
is_std_separator, used in chunk 79b.
separator_found, never used.
separator_not_found, never used.
Uses SCRATCH2 209 and get_dictionary_addr 94.

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.

85 ⟨*ASCII to Zchar* 85⟩≡ (213) 86a ▷

```
ascii_to_zchar:
    SUBROUTINE

    STOB     #$00, LOCKED_ALPHABET
    LDX      #$00
    LDY      #$06

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

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

Defines:
ascii_to_zchar, used in chunk 82.
Uses LOCKED_ALPHABET 209, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, STOB 13b, and ZCHAR_SCRATCH2 209.

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.

86a ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁85 86b▷

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

Uses SCRATCH2 209, SCRATCH3 209, and ZCHAR_SCRATCH1 209.

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.

86b ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁86a 87b▷

```
  .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 209, SCRATCH1 209, SCRATCH3 209, and get_alphabet_for_char 87a.

87a ⟨*Get alphabet for char* 87a⟩≡ (213)

```
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 86b, 87b, and 91a.

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.

87b ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁86b 88a▷

```
    LDX     SCRATCH2
    LDA     ZCHAR_SCRATCH1,X
    JSR     get_alphabet_for_char
    CMP     SCRATCH1
    BNE     .shift_alphabet
```

Uses SCRATCH1 209, SCRATCH2 209, ZCHAR_SCRATCH1 209, and get_alphabet_for_char 87a.

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.

88a ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁87b 88b▷

```
    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_3 107, LOCKED_ALPHABET 209, MOVB 13b, and SCRATCH1 209.

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

88b ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁88a 88c▷

```
    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 209 and ZCHAR_SCRATCH2 209.

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.

88c ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁88b 90▷

```
    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 209, SCRATCH3 209, and z_compress 89.

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

89 ⟨*Z compress* 89⟩≡ (213)

```
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 88c, 90, 91b, and 93.

Uses `ZCHAR_SCRATCH2` 209.

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.

90 ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁88c 91a▷

```
.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_3 107, LOCKED_ALPHABET 209, SCRATCH1 209, SCRATCH3 209, ZCHAR_SCRATCH2 209, and z_compress 89.

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

91a ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁90 91b▷

```
  .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` 209, `SCRATCH3` 209, and `get_alphabet_for_char` 87a.

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.

91b ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁91a 91c▷

```
  .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` 209, `SCRATCH3` 209, `ZCHAR_SCRATCH2` 209, and `z_compress` 89.

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.

91c ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁91b 92a▷

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

Uses `SCRATCH3` 209.

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.

92a ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁91c 93▷

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

Uses `SCRATCH3` 209 and `search_nonalpha_table` 92b.

92b ⟨*Search nonalpha table* 92b⟩≡ (213)

```
  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 92a.
Uses `a2_table` 71a.

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.

93 ⟨*ASCII to Zchar* 85⟩+≡ (213) ◁92a

```
    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 209, SCRATCH3 209, ZCHAR_SCRATCH2 209, and z_compress 89.

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`.

94 ⟨*Get dictionary address 94*⟩≡ (213)

```
get_dictionary_addr:
    SUBROUTINE

    LDY     #HEADER_DICT_ADDR
    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 84 and 95.

Uses `ADDW` 18a, `HEADER_DICT_ADDR` 211a, and `SCRATCH2` 209.

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.

95 ⟨*Match dictionary word* 95⟩≡ (213) 96a ▷

```
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 82.
Uses ADDA 16a, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, and get_dictionary_addr 94.

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.

96a ⟨*Match dictionary word* 95⟩+≡ (213) ◁95 96b▷

```
.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 17b, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, and ZCHAR_SCRATCH2 209.

96b ⟨*Match dictionary word* 95⟩+≡ (213) ◁96a 97▷

```
.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 17b, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, and SUBB2 19b.

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.

97 ⟨*Match dictionary word* 95⟩+≡ (213) ◁96b 98a▷

```
.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 17b, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, SUBB 19a, and ZCHAR_SCRATCH2 209.

If the search failed, we return 0 in SCRATCH1.

98a ⟨*Match dictionary word* 95⟩+≡ (213) ◁97 98b▷

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

Uses SCRATCH1 209.

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

98b ⟨*Match dictionary word* 95⟩+≡ (213) ◁98a

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

Uses SCRATCH1 209, SCRATCH2 209, and SUBW 20a.

Chapter 9

Arithmetic routines

9.1 Negation and sign manipulation

`negate` negates the word in `SCRATCH2`.

99 ⟨*negate* 99⟩≡ (213)

```
  negate:
      SUBROUTINE

      SUBWL    #$0000, SCRATCH2, SCRATCH2
      RTS
```

Defines:
  `negate`, used in chunks 100a, 101, and 109.
Uses `SCRATCH2` 209 and `SUBWL` 20b.

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.

100a ⟨*Flip sign* 100a⟩≡ (213)

```
flip_sign:
    SUBROUTINE

    ORA     #$00
    BMI     .do_negate
    RTS

.do_negate:
    INC     SIGN_BIT
    JMP     negate
```

Defines:
flip_sign, used in chunk 100b.
Uses negate 99.

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.

100b ⟨*Check sign* 100b⟩≡ (213)

```
check_sign:
    SUBROUTINE

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

Defines:
check_sign, used in chunks 175–77.
Uses SCRATCH1 209, SCRATCH2 209, STOB 13b, and flip_sign 100a.

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.

101 ⟨*Set sign* 101⟩≡ (213)

```
set_sign:
    SUBROUTINE

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

Defines:
set_sign, used in chunk 177.

Uses negate 99.

9.2 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.

102 ⟨*mulu16* 102⟩≡ (213)

```
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 177.

Uses `ADDWC` 18b, `MOVW` 14a, `PSHW` 14b, `PULW` 15a, `RORW` 21b, `SCRATCH1` 209, `SCRATCH2` 209, `SCRATCH3` 209, and `STOW` 12.

9.3 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:

104 ⟨*trace of divu16* 104⟩≡

```
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.

105 ⟨*divu16* 105⟩≡ (213)

```
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 108, 175, 176, and 178a.

Uses MOVW 14a, PSHW 14b, PULW 15a, ROLW 21a, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, and STOW 12.

9.4 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.

106a ⟨*cmpu16* 106a⟩≡ (213)

```
cmpu16:
    SUBROUTINE

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

Defines:
cmpu16, used in chunks 106b and 185a.
Uses SCRATCH1 209 and SCRATCH2 209.

cmp16 compares the two signed words in SCRATCH1 and SCRATCH2.

106b ⟨*cmp16* 106b⟩≡ (213)

```
cmp16:
    SUBROUTINE

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

Defines:
cmp16, used in chunks 181a, 183a, and 184a.
Uses SCRATCH1 209, SCRATCH2 209, and cmpu16 106a.

9.5 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.

107 ⟨*A mod 3* 107⟩≡ (213)

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

.end:
    RTS
```

Defines:
A_mod_3, used in chunks 68, 88a, and 90.

9.6 Printing numbers

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

108 ⟨*Print number* 108⟩≡ (213)

```
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 73 and 190a.

Uses SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, STOB 13b, STOW 12, buffer_char 61, divu16 105, and print_negative_num 109.

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.

109 ⟨*Print negative number* 109⟩≡ (213)

```
print_negative_num:
    SUBROUTINE

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

Defines:
`print_negative_num`, used in chunk 108.

Uses `buffer_char` 61 and `negate` 99.

Chapter 10

Disk routines

110 ⟨*iob struct* 110⟩≡ (212c)

```
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      #$01            ; last_drive_number

dct:
    DC      #$00            ; device_type (0 for DISK II)
    DC      #$01            ; phases_per_track (1 for DISK II)
dct.motor_count:
    DC.W    #$D8EF          ; motor_on_time_count ($EFD8 for DISK II)
```

Defines:
`dct`, used in chunk 113.
`iob`, used in chunks 111, 150, and 152.

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.

111 ⟨*Do RWTS on sector* 111⟩≡ (212c)

```
do_rwts_on_sector:
    SUBROUTINE

    STA     iob.command
    MOVW    SCRATCH2, iob.buffer
    STOB    #$03, 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 112 and 113.
Uses MOVW 14a, RWTS 209, SCRATCH1 209, SCRATCH2 209, SECTORS_PER_TRACK 209, STOB 13b, and iob 110.

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.

112 ⟨*Reading sectors* 112⟩≡ (212c)

```
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 158b.
read_from_sector, used in chunks 36c, 38a, 47, and 50.
read_next_sector, used in chunks 156c and 158a.

Uses BUFF_AREA 209, INCW 15b, SCRATCH1 209, SCRATCH2 209, STOW 12, and do_rwts_on_sector 111.

For some reason the `write_next_sector` routine temporarily stores the standard `#$D8EF` into the disk motor on-time count. There doesn't seem to be any reason for this, since the motor count is never set to anything else.

113 ⟨*Writing sectors* 113⟩≡ (212c)

```
write_next_sector:
    SUBROUTINE

    STOW     #BUFF_AREA, SCRATCH2

inc_sector_and_write:
    SUBROUTINE

    INCW     SCRATCH1

.write_next_sector:
    PSHW     dct.motor_count
    STOW2    #$D8EF, dct.motor_count
    LDA      #$02
    JSR      do_rwts_on_sector
    PULW     dct.motor_count
    RTS
```

Defines:
`inc_sector_and_write`, used in chunk 155b.
`write_next_sector`, used in chunks 154b and 155a.

Uses `BUFF_AREA` 209, `INCW` 15b, `PSHW` 14b, `PULW` 15a, `SCRATCH1` 209, `SCRATCH2` 209, `STOW` 12, `STOW2` 13a, `dct` 110, and `do_rwts_on_sector` 111.

Chapter 11

The instruction dispatcher

11.1 Executing an instruction

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

114 ⟨*Instruction tables* 114⟩≡ (212b)

```
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 118b.
`routines_table_1op`, used in chunk 120b.
`routines_table_2op`, used in chunk 122c.

routines_table_var, used in chunk 124.
Uses illegal_opcode 162, instr_add 174b, instr_and 179a, instr_call 130, instr_clear_attr 191, instr_dec 174a, instr_dec_chk 180b, instr_div 175, instr_get_next_prop 193, instr_get_parent 194, instr_get_prop 195, instr_get_prop_addr 198, instr_get_prop_len 199, instr_get_sibling 200, instr_inc 173c, instr_inc_chk 181a, instr_insert_obj 201, instr_je 181b, instr_jg 183a, instr_jin 183b, instr_jl 184a, instr_jump 186a, instr_jz 184b, instr_load 169a, instr_loadb 170a, instr_loadw 169b, instr_mod 176, instr_mul 177, instr_new_line 188b, instr_nop 204a, instr_not 179b, instr_or 180a, instr_pop 172b, instr_print 189a, instr_print_addr 189b, instr_print_char 189c, instr_print_num 190a, instr_print_obj 190b, instr_print_paddr 190c, instr_print_ret 186b, instr_pull 173a, instr_push 173b, instr_put_prop 202, instr_quit 205, instr_random 178a, instr_remove_obj 203a, instr_restart 204b, instr_restore 156b, instr_ret 134, instr_ret_popped 187a, instr_rfalse 187b, instr_rtrue 188a, instr_save 153a, instr_set_attr 203b, instr_sread 77, instr_store 170b, instr_storeb 172a, instr_storew 171, instr_sub 178c, instr_test 185a, and instr_test_attr 185b.

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.

116 ⟨*Execute instruction* 116⟩≡ (212b)
```
.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 209, SCRATCH1 209, and SCRATCH2 209.

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.

11.2 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`.

117 ⟨*Do instruction* 117⟩≡ (212b) 118a ▷

```
do_instruction:
    SUBROUTINE

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

Defines:
`do_instruction`, used in chunks 42b, 78, 133b, 163, 165b, 168, 170–74, 188–91, and 201–4.
Uses `CURR_OPCODE` 209, `MOVB` 13b, `MOVW` 14a, `OPERAND_COUNT` 209, `STOB` 13b, `TMP_Z_PC` 209, `Z_PC` 209, and `get_next_code_byte` 45.

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

11.3 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.

118a ⟨*Do instruction* 117⟩+≡ (212b) ◁117

```
    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 123⟩
```

Uses `get_next_code_byte` 45.

11.3.1 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.

118b ⟨*Handle 0op instructions* 118b⟩≡ (212b)

```
.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` 209, `STOW` 12, `illegal_opcode` 162, and `routines_table_0op` 114.

11.3.2 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.

119a ⟨*Handle 1op instructions* 119a⟩≡ (212b) 119b ▷

```
.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` 125b.

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

119b ⟨*Handle 1op instructions* 119a⟩+≡ (212b) ◁ 119a 119c ▷

```
.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` 125a.

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

119c ⟨*Handle 1op instructions* 119a⟩+≡ (212b) ◁ 119b 119d ▷

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

Uses `get_var_content` 126.

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

119d ⟨*Handle 1op instructions* 119a⟩+≡ (212b) ◁ 119c 120a ▷

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

Uses `MOVW` 14a, `OPERAND0` 209, `OPERAND_COUNT` 209, `SCRATCH2` 209, and `STOB` 13b.

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.

120a ⟨*Handle 1op instructions* 119a⟩+≡ (212b) ◁119d 120b▷

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

Uses CURR_OPCODE 209 and illegal_opcode 162.

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

120b ⟨*Handle 1op instructions* 119a⟩+≡ (212b) ◁120a

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

Uses SCRATCH2 209, STOW 12, and routines_table_1op 114.

11.3.3 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`.

121a ⟨*Handle 2op instructions* 121a⟩≡ (212b) 121b ▷

```
.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` 14a, `OPERAND0` 209, `SCRATCH2` 209, `get_const_byte` 125a, and `get_var_content` 126.

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`.

121b ⟨*Handle 2op instructions* 121a⟩+≡ (212b) ◁ 121a 122a ▷

```
    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` 209, `MOVW` 14a, `OPERAND1` 209, `SCRATCH2` 209, `get_const_byte` 125a, and `get_var_content` 126.

OPERAND_COUNT is set to 2.

122a ⟨*Handle 2op instructions* 121a⟩+≡ (212b) ◁121b 122b▷
```
    STOB      #$02, OPERAND_COUNT
```
Uses OPERAND_COUNT 209 and STOB 13b.

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

122b ⟨*Handle 2op instructions* 121a⟩+≡ (212b) ◁122a 122c▷
```
    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 209 and illegal_opcode 162.

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

122c ⟨*Handle 2op instructions* 121a⟩+≡ (212b) ◁122b
```
.go_to_op2:
    PHA
    STOW      routines_table_2op, SCRATCH2
    PLA
    JMP       .opcode_table_jump
```
Uses SCRATCH2 209, STOW 12, and routines_table_2op 114.

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

11.3.4 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`.

123 ⟨*Handle varop instructions* 123⟩≡ (118a) 124▷

```
    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+1,X
    INX
    INX
    INC      OPERAND_COUNT
    PLA                                ; shift operand map left 2 bits
    SEC
    ROL
    SEC
    ROL
    JMP      .get_next_operand
```

Uses OPERAND0 209, OPERAND_COUNT 209, SCRATCH2 209, get_const_byte 125a, get_const_word 125b, and get_var_content 126.

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.

124 ⟨*Handle varop instructions* 123⟩+≡ (118a) ◁123

```
  .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 209, SCRATCH2 209, STOW 12, illegal_opcode 162, and routines_table_var 114.

11.4 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.

125a ⟨*Get const byte* 125a⟩≡ (212b)

```
get_const_byte:
    SUBROUTINE

    JSR     get_next_code_byte
    STA     SCRATCH2
    STOB    #$00, SCRATCH2+1
    RTS
```

Defines:
get_const_byte, used in chunks 119b, 121, and 123.
Uses SCRATCH2 209, STOB 13b, and get_next_code_byte 45.

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.

125b ⟨*Get const word* 125b⟩≡ (212b)

```
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 119a and 123.
Uses SCRATCH2 209 and get_next_code_byte 45.

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.

126 ⟨*Get var content* 126⟩≡ (212b)

```
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:
    ; var_ptr += GLOBAL_ZVARS_ADDR
    ADDW    GLOBAL_ZVARS_ADDR, SCRATCH1, SCRATCH1
```

```
  .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 127.
  get_top_of_stack, never used.
  get_var_content, used in chunks 119c, 121, and 123.
Uses ADDW 18a, GLOBAL_ZVARS_ADDR 209, LOCAL_ZVARS 209, SCRATCH1 209, SCRATCH2 209, Z_PC 209, get_next_code_byte 45, and pop 44.

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

127 ⟨*Get var content in A* 127⟩≡ (212b)

```
  var_get:
      SUBROUTINE

      ORA       #$00
      BEQ       pop_push
      JMP       get_nonstack_var
```

Defines:
  var_get, used in chunks 73, 164, and 169a.
Uses get_nonstack_var 126 and pop_push 129.

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.

128 ⟨*Store var* 128⟩≡ (212b)

```
store_var:
    SUBROUTINE

    PSHW     SCRATCH2             ; A = get_next_code_byte()
    JSR      get_next_code_byte
    TAX
    PULW     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
    ADDW     GLOBAL_ZVARS_ADDR, SCRATCH1, SCRATCH1
    LDY      #$00
    LDA      SCRATCH2+1
    STA      (SCRATCH1),Y
    INY
    LDA      SCRATCH2
```

```
    STA     (SCRATCH1),Y
    RTS
```

Defines:
  `store_var`, used in chunks 163a and 192.
Uses `ADDW` 18a, `GLOBAL_ZVARS_ADDR` 209, `LOCAL_ZVARS` 209, `PSHW` 14b, `PULW` 15a, `SCRATCH1` 209, `SCRATCH2` 209, `get_next_code_byte` 45, and `push` 43.

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.

129 ⟨*Store to var A* 129⟩≡ (212b)

```
var_put:
    SUBROUTINE

    ORA     #$00
    BEQ     .pop_push
    JMP     store_var2

pop_push:
    JSR     pop
    JMP     push

.pop_push:
    PSHW    SCRATCH2
    JSR     pop
    PULW    SCRATCH2
    JMP     push
```

Defines:
  `pop_push`, used in chunk 127.
  `var_put`, used in chunks 164a and 170b.
Uses `PSHW` 14b, `PULW` 15a, `SCRATCH2` 209, `pop` 44, and `push` 43.

Chapter 12

Calls and returns

12.1 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.

130 ⟨*Instruction call* 130⟩≡ (213) 131a ▷

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

Defines:
`instr_call`, used in chunk 114.

Uses `OPERAND0` 209, `SCRATCH2` 209, `STOW` 12, and `store_and_next` 163a.

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.

131a ⟨*Instruction call* 130⟩+≡ (213) ◁130 131b▷

```
  .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 209, FRAME_Z_SP 209, MOVB 13b, MOVW 14a, SCRATCH2 209, STOB 13b, ZCODE_PAGE_VALID 209, Z_PC 209, and push 43.

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

131b ⟨*Instruction call* 130⟩+≡ (213) ◁131a 131c▷

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

Uses OPERAND0 209 and Z_PC 209.

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

131c ⟨*Instruction call* 130⟩+≡ (213) ◁131b 132▷

```
      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 45.

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

132 ⟨*Instruction call* 130⟩+≡ (213) ◁131c 133a▷

```
    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 209, SCRATCH2 209, get_next_code_byte 45, and push 43.

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

133a ⟨*Instruction call* 130⟩+≡ (213) ◁132 133b▷

```
.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] = OPERANDS[operand+1]
    LDA      OPERAND1+1,X            ; high byte first
    LDX      SCRATCH2
    STA      LOCAL_ZVARS,X
    INC      SCRATCH2
    LDX      SCRATCH1
    LDA      OPERAND1,X
    LDX      SCRATCH2
    STA      LOCAL_ZVARS,X
    INC      SCRATCH2                ; ++zvar
    INC      SCRATCH1                ; ++operand
    INC      SCRATCH1
    DEC      SCRATCH3                ; --count
    BNE      .loop                   ; if (count) .loop
```

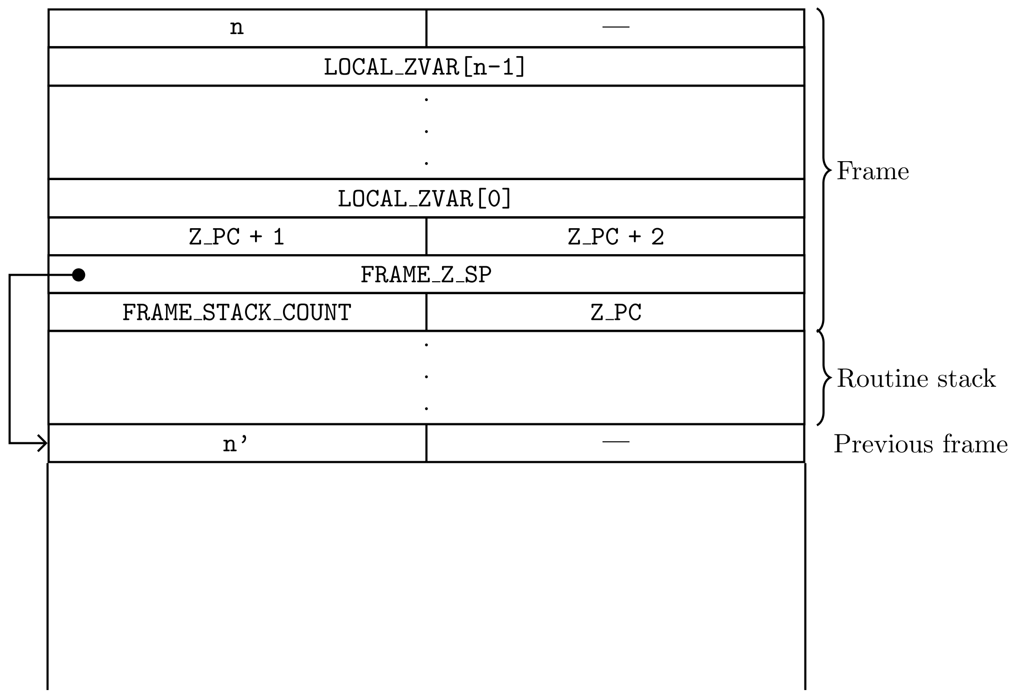
Uses LOCAL_ZVARS 209, OPERAND1 209, OPERAND_COUNT 209, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, and STOB 13b.

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.

133b ⟨*Instruction call* 130⟩+≡ (213) ◁133a

```
.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 209, FRAME_Z_SP 209, MOVB 13b, MOVW 14a, PULB 14c, SCRATCH2 209, STACK_COUNT 209, Z_SP 209, do_instruction 117, and push 43.



12.2 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.

134 ⟨*Instruction ret* 134⟩≡ (213) 135a▷

```
  instr_ret:
      SUBROUTINE

      MOVW     FRAME_Z_SP, Z_SP
      MOVB     FRAME_STACK_COUNT, STACK_COUNT
```

Defines:
`instr_ret`, used in chunks 114, 187a, and 188a.

Uses `FRAME_STACK_COUNT` 209, `FRAME_Z_SP` 209, `MOVB` 13b, `MOVW` 14a, `STACK_COUNT` 209, and `Z_SP` 209.

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.

135a ⟨*Instruction ret* 134⟩+≡ (213) ◁134 135b▷

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

Uses SCRATCH2 209 and pop 44.

We then set up the loop variables for restoring the locals. I'm not really sure why we start with GLOBAL_ZVARS_ADDR.

135b ⟨*Instruction ret* 134⟩+≡ (213) ◁135a 135c▷

```
    STOW    GLOBAL_ZVARS_ADDR, SCRATCH1     ; ptr = GLOBAL_ZVARS_ADDR
    MOVB    SCRATCH2, SCRATCH3              ; count = STRATCH2
    ASL                                     ; ptr += 2 * count
    ADDA    SCRATCH1
```

Uses ADDA 16a, GLOBAL_ZVARS_ADDR 209, MOVB 13b, SCRATCH1 209, SCRATCH2 209, SCRATCH3 209, and STOW 12.

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

135c ⟨*Instruction ret* 134⟩+≡ (213) ◁135b 135d▷

```
.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 209, SCRATCH2 209, SCRATCH3 209, SUBB 19a, and pop 44.

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

135d ⟨*Instruction ret* 134⟩+≡ (213) ◁135c 136▷

```
.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 209, FRAME_Z_SP 209, MOVB 13b, MOVW 14a, SCRATCH2 209, Z_PC 209, and pop 44.

Finally, we store the return value.

136 ⟨*Instruction ret* 134⟩+≡ (213) ◁135d

```
    STOB    #$00, ZCODE_PAGE_VALID
    MOVW    OPERAND0, SCRATCH2
    JMP     store_and_next
```

Uses MOVW 14a, OPERAND0 209, SCRATCH2 209, STOB 13b, ZCODE_PAGE_VALID 209, and store_and_next 163a.

Chapter 13

Objects

13.1 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).

13.2 Getting an object's address

The `get_object_addr` 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).

137 ⟨*Get object address* 137⟩≡ (213) 138a ▷

```
get_object_addr:
    SUBROUTINE

    STA     SCRATCH2
    STOB    #$00, 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
```

Defines:
`get_object_addr`, used in chunks 139–42, 144, 183b, 192, 194, 200, and 201.
Uses `SCRATCH2` 209 and `STOB` 13b.

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.

138a ⟨*Get object address* 137⟩+≡ (213) ◁137 138b▷

```
.continue:
    ADC     #FIRST_OBJECT_OFFSET
    STA     SCRATCH2
    BCC     .continue2
    INC     SCRATCH2+1
```

Uses `FIRST_OBJECT_OFFSET` 211a and `SCRATCH2` 209.

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`.

138b ⟨*Get object address* 137⟩+≡ (213) ◁138a

```
.continue2:
    LDY     #HEADER_OBJECT_TABLE_ADDR+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` 211a and `SCRATCH2` 209.

13.3 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.

139a ⟨*Remove object* 139a⟩≡ (213) 139b ▷

```
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 201 and 203a.
Uses `OBJECT_PARENT_OFFSET` 211a, `OPERAND0` 209, `SCRATCH2` 209, and `get_object_addr` 137.

Next, we save the object's address on the stack.

139b ⟨*Remove object* 139a⟩+≡ (213) ◁ 139a 139c ▷

```
    TAX                           ; save obj_ptr
    PSHW    SCRATCH2
    TXA
```

Uses `PSHW` 14b and `SCRATCH2` 209.

Next, we get the parent's first child pointer.

139c ⟨*Remove object* 139a⟩+≡ (213) ◁ 139b 140a ▷

```
    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` 211a, `SCRATCH2` 209, and `get_object_addr` 137.

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.

140a ⟨*Remove object* 139a⟩+≡ (213) ◁139c 140b▷

```
        CMP     OPERAND0                ; if (child_num != obj_num) loop
        BNE     .loop
```

Uses OPERAND0 209.

But otherwise, we get the object's sibling and replace the parent's first child with it.

140b ⟨*Remove object* 139a⟩+≡ (213) ◁140a 140d▷

```
        PULW    SCRATCH1                ; restore obj_ptr
        PSHW    SCRATCH1
        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 211a, OBJECT_SIBLING_OFFSET 211a, PSHW 14b, PULW 15a, SCRATCH1 209, and SCRATCH2 209.

Detaching the object means we null out the parent pointer of the object. Then we can return.

140c ⟨*Detach object* 140c⟩≡ (141a)

```
    .detach:
        PULW    SCRATCH2                ; restore obj_ptr
        LDY     #OBJECT_PARENT_OFFSET  ; obj_ptr->parent = 0
        LDA     #$00
        STA     (SCRATCH2),Y
        INY
        STA     (SCRATCH2),Y
        RTS
```

Uses OBJECT_PARENT_OFFSET 211a, PULW 15a, and SCRATCH2 209.

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.

140d ⟨*Remove object* 139a⟩+≡ (213) ◁140b 141a▷

```
    .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 211a, OPERAND0 209, SCRATCH2 209, and get_object_addr 137.

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.

141a ⟨*Remove object* 139a⟩+≡ (213) ◁140d

```
    PULW    SCRATCH1              ; restore obj_ptr
    PSHW    SCRATCH1
    LDA     (SCRATCH1),Y          ; child_ptr->next = obj_ptr->next
    STA     (SCRATCH2),Y

⟨Detach object 140c⟩
```

Uses PSHW 14b, PULW 15a, SCRATCH1 209, and SCRATCH2 209.

13.4 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.

141b ⟨*Print object in A* 141b⟩≡ (213)

```
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 73 and 190b.

Uses INCW 15b, MOVW 14a, OBJECT_PROPS_OFFSET 211a, SCRATCH1 209, SCRATCH2 209, get_object_addr 137, load_address 52b, and print_zstring 66.

13.5 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.

142 ⟨*Get attribute pointer and mask* 142⟩≡ (213) 143a ▷

```
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 185b, 191, and 203b.

Uses `INCW` 15b, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, and `get_object_addr` 137.

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.

143a ⟨*Get attribute pointer and mask* 142⟩+≡ (213) ◁142 143b▷

```
    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 209, SCRATCH3 209, and STOW 12.

Finally, we load the attribute word into SCRATCH1.

143b ⟨*Get attribute pointer and mask* 142⟩+≡ (213) ◁143a

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

Uses SCRATCH1 209 and SCRATCH2 209.

13.6 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.

144 ⟨*Get property pointer* 144⟩≡ (213)

```
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 193, 195, 198, and 202.
Uses `ADDW` 18a, `OBJECT_PROPS_OFFSET` 211a, `OPERAND0` 209, `SCRATCH1` 209, `SCRATCH2` 209, and `get_object_addr` 137.

The `get_property_num` routine gets the property number being currently pointed to.

145a ⟨*Get property number* 145a⟩≡ (213)
```
  get_property_num:
      SUBROUTINE

      LDA      (SCRATCH2),Y
      AND      #$1F
      RTS
```
Defines:
  `get_property_num`, used in chunks 193, 195, 198, and 202.
Uses `SCRATCH2` 209.

The `get_property_len` routine gets the length of the property being currently pointed to, minus one.

145b ⟨*Get property length* 145b⟩≡ (213)
```
  get_property_len:
      SUBROUTINE

      LDA      (SCRATCH2),Y
      ROR
      ROR
      ROR
      ROR
      ROR
      AND      #$07
      RTS
```
Defines:
  `get_property_len`, used in chunks 146, 197, 199, and 202.
Uses `SCRATCH2` 209.

The `next_property` routine updates the `Y` register to point to the next property in the property table.

146 ⟨*Next property* 146⟩≡ (213)

```
next_property:
    SUBROUTINE

    JSR     get_property_len
    TAX

.loop:
    INY
    DEX
    BPL     .loop
    INY
    RTS
```

Defines:
`next_property`, used in chunks 193, 195, 198, and 202.

Uses `get_property_len` 145b.

Chapter 14

Saving and restoring the game

14.0.1 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:

147 ⟨*Insert save diskette* 147⟩≡ (212c) 148a ▷

```
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 153a and 156b.

Uses `SCRATCH2` 209, `STOW` 12, `dump_buffer_with_more` 59, `home` 55a, `print_ascii_string` 63b, and `sPleaseInsert` 148b.

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.

148a ⟨*Insert save diskette* 147⟩+≡ (212c) ◁147 150a▷

```
.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` 61, `prompt_offset` 148b, `sPositionPrompt` 148b, `sSlotPrompt` 148b, and `save_position` 148b.

148b ⟨*Save diskette strings* 148b⟩≡ (212c)

```
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 148–50.
- `sDrivePrompt`, used in chunk 150b.
- `sPleaseInsert`, used in chunk 147.
- `sPositionPrompt`, used in chunk 148a.
- `sReturnToBegin`, used in chunk 151a.
- `sSlotPrompt`, used in chunks 148–50.
- `save_drive`, used in chunks 150b and 236.
- `save_position`, used in chunks 148a and 151b.
- `save_slot`, used in chunks 149 and 150a.

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.

149 ⟨*Get prompted number from user* 149⟩≡ (212c)

```
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 17a, CH 208, CLREOL 208, INVFLG 208, RDKEY 208, SCRATCH2 209, STOW 12, cout_string 54, dump_buffer_line 58, dump_buffer_with_more 59, print_ascii_string 63b, prompt_offset 148b, sSlotPrompt 148b, and save_slot 148b.

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.

150a ⟨*Insert save diskette* 147⟩+≡ (212c) ◁148a 150b▷

```
.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 61, iob 110, prompt_offset 148b, sSlotPrompt 148b, and save_slot 148b.

Next, we ask the user for the drive number, which must be 1 or 2. This value is stored in iob.drive.

150b ⟨*Insert save diskette* 147⟩+≡ (212c) ◁150a 151a▷

```
.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 61, iob 110, prompt_offset 148b, sDrivePrompt 148b, sSlotPrompt 148b, and save_drive 148b.

Next, we prompt the user to start.

151a ⟨*Insert save diskette* 147⟩+≡ (212c) ◁150b 151b▷

```
  .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 208, SCRATCH2 209, STOW 12, dump_buffer_line 58, dump_buffer_with_more 59, print_ascii_string 63b, and sReturnToBegin 148b.

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.

151b ⟨*Insert save diskette* 147⟩+≡ (212c) ◁151a

```
      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 17a, SCRATCH1 209, dump_buffer_with_more 59, and save_position 148b.

When the save is eventually complete, the user is prompted to reinsert the game diskette.

152 ⟨*Reinsert game diskette* 152⟩≡ (212c)

```
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:
    STOB    #$60, iob.slot_times_16
    STOB    #$01, iob.drive
    RTS
```

Defines:
`please_reinsert_game_diskette`, used in chunks 155c, 156a, and 159.
`sPressReturnToContinue`, never used.
`sReinsertGameDiskette`, never used.

Uses `RDKEY` 208, `SCRATCH2` 209, `STOB` 13b, `STOW` 12, `dump_buffer_line` 58, `dump_buffer_with_more` 59, `iob` 110, and `print_ascii_string` 63b.

14.0.2 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.

153a ⟨*Instruction save* 153a⟩≡ (212c) 153b ▷

```
instr_save:
    SUBROUTINE

    JSR      please_insert_save_diskette
```

Defines:
`instr_save`, used in chunk 114.
Uses `please_insert_save_diskette` 147.

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.

153b ⟨*Instruction save* 153a⟩+≡ (212c) ◁ 153a 153c ▷

```
    LDX      #$00
    LDY      #HEADER_VERSION
    LDA      (Z_HEADER_ADDR),Y
    STA      BUFF_AREA,X
    INX
```

Uses `BUFF_AREA` 209.

Next, we copy the 3 bytes of `Z_PC` to the buffer. This is actually done in reverse order.

153c ⟨*Instruction save* 153a⟩+≡ (212c) ◁ 153b 154b ▷

```
    STOW     #Z_PC, SCRATCH2
    LDY      #$03
    JSR      copy_data_to_buff
```

Uses `SCRATCH2` 209, `STOW` 12, `Z_PC` 209, and `copy_data_to_buff` 154a.

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.

154a ⟨*Copy data to buff* 154a⟩≡ (212c)

```
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 153–55.
Uses BUFF_AREA 209 and SCRATCH2 209.

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.

154b ⟨*Instruction save* 153a⟩+≡ (212c) ◁153c 155a▷

```
    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 209, SCRATCH2 209, STACK_COUNT 209, STOW 12, copy_data_to_buff 154a, and write_next_sector 113.

The second sector written contains 256 bytes starting from #$0280, and the third sector contains 256 bytes starting from #$0380.

155a ⟨*Instruction save* 153a⟩+≡ (212c) ◁154b 155b▷

```
        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 209, STOW 12, copy_data_to_buff 154a, and write_next_sector 113.

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.

155b ⟨*Instruction save* 153a⟩+≡ (212c) ◁155a 155c▷

```
        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 211a, MOVW 14a, SCRATCH2 209, SCRATCH3 209, and inc_sector_and_write 113.

Finally, we ask the user to reinsert the game diskette, and we're done. The instruction branches, assuming success.

155c ⟨*Instruction save* 153a⟩+≡ (212c) ◁155b 156a▷

```
        JSR     please_reinsert_game_diskette
        JMP     branch
```

Uses branch 165a and please_reinsert_game_diskette 152.

On failure, the instruction also asks the user to reinsert the game diskette, but branches assuming failure.

156a ⟨*Instruction save* 153a⟩+≡ (212c) ◁155c

```
  .fail:
      JSR      please_reinsert_game_diskette
      JMP      negated_branch
```

Uses negated_branch 165a and please_reinsert_game_diskette 152.

14.0.3 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.

156b ⟨*Instruction restore* 156b⟩≡ (212c) 156c▷

```
  instr_restore:
      SUBROUTINE

      JSR      please_insert_save_diskette
```

Defines:
instr_restore, used in chunk 114.
Uses please_insert_save_diskette 147.

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.

156c ⟨*Instruction restore* 156b⟩+≡ (212c) ◁156b 157a▷

```
      JSR      read_next_sector
      BCC      .continue
      JMP      .fail

  .continue:
      LDX      #$00
      LDY      #HEADER_VERSION
      LDA      (Z_HEADER_ADDR),Y
      CMP      BUFF_AREA,X
      BEQ      .continue2
      JMP      .fail
```

Uses BUFF_AREA 209 and read_next_sector 112.

We also save the current game flags in the header at byte #$11.

157a ⟨*Instruction restore* 156b⟩+≡ (212c) ◁156c 157b▷

```
.continue2:
    LDY      #HEADER_FLAGS2+1
    LDA      (Z_HEADER_ADDR),Y
    STA      SIGN_BIT
```

Uses HEADER_FLAGS2 211a.

We then restore the Z_PC, local variables, and stack state from the same sector.

157b ⟨*Instruction restore* 156b⟩+≡ (212c) ◁157a 158a▷

```
    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 209, SCRATCH2 209, STACK_COUNT 209, STOW 12, ZCODE_PAGE_VALID 209, Z_PC 209, and copy_data_from_buff 157c.

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.

157c ⟨*Copy data from buff* 157c⟩≡ (212c)

```
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 157b and 158a.
Uses BUFF_AREA 209 and SCRATCH2 209.

Next we restore 256 bytes starting from #$0280 from the second sector, and 256 bytes starting from #$0380 from the third sector.

158a ⟨*Instruction restore* 156b⟩+≡ (212c) ◁157b 158b▷

```
        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 209, STOW 12, copy_data_from_buff 157c, and read_next_sector 112.

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.

158b ⟨*Instruction restore* 156b⟩+≡ (212c) ◁158a 158c▷

```
        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_read
        BCS     .fail
        INC     SCRATCH2+1
        DEC     SCRATCH3
        BNE     .loop
```

Uses HEADER_STATIC_MEM_BASE 211a, MOVW 14a, SCRATCH2 209, SCRATCH3 209, and inc_sector_and_read 112.

Then we restore the game flags in the header at byte #$11 from before the actual restore.

158c ⟨*Instruction restore* 156b⟩+≡ (212c) ◁158b 159a▷

```
        LDA     SIGN_BIT
        LDY     #HEADER_FLAGS2+1
        STA     (Z_HEADER_ADDR),Y
```

Uses HEADER_FLAGS2 211a.

Finally, we ask the user to reinsert the game diskette, and we're done. The instruction branches, assuming success.

159a ⟨*Instruction restore* 156b⟩+≡ (212c) ◁158c 159b▷

```
        JSR     please_reinsert_game_diskette
        JMP     branch
```

Uses `branch` 165a and `please_reinsert_game_diskette` 152.

On failure, the instruction also asks the user to reinsert the game diskette, but branches assuming failure.

159b ⟨*Instruction restore* 156b⟩+≡ (212c) ◁159a

```
.fail:
        JSR     please_reinsert_game_diskette
        JMP     negated_branch
```

Uses `negated_branch` 165a and `please_reinsert_game_diskette` 152.

Chapter 15

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

15.1 Instruction utilities

There are a few utilities that are used in common by instructions.

illegal_opcode hits a BRK instruction.

162 ⟨*Instruction illegal opcode* 162⟩≡ (213)

```
illegal_opcode:
    SUBROUTINE

    JSR     brk
```

Defines:
illegal_opcode, used in chunks 114, 118b, 120a, 122b, and 124.
Uses brk 39b.

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.

163a ⟨*Store and go to next instruction* 163a⟩≡ (212b)

```
store_zero_and_next:
    SUBROUTINE

    LDA     #$00

store_A_and_next:
    SUBROUTINE

    STA     SCRATCH2
    STOB    #$00, SCRATCH2+1

store_and_next:
    SUBROUTINE

    JSR     store_var
    JMP     do_instruction
```

Defines:
  store_A_and_next, used in chunks 193 and 199.
  store_and_next, used in chunks 130, 136, 169, 170a, 174b, 176–80, 194, and 196–98.
  store_zero_and_next, used in chunks 193 and 198.
Uses SCRATCH2 209, STOB 13b, do_instruction 117, and store_var 128.

The print_zstring_and_next routine prints the z-encoded string at Z_PC2 to the screen, and then goes to the next instruction.

163b ⟨*Print zstring and go to next instruction* 163b⟩≡ (213)

```
print_zstring_and_next:
    SUBROUTINE

    JSR     print_zstring
    JMP     do_instruction
```

Defines:
  print_zstring_and_next, used in chunks 189b and 190c.
Uses do_instruction 117 and print_zstring 66.

The inc_var routine increments the variable in OPERAND0, and also stores the result in SCRATCH2.

164a ⟨*Increment variable* 164a⟩≡ (213)

```
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 173c and 181a.
Uses INCW 15b, OPERAND0 209, PSHW 14b, PULW 15a, SCRATCH2 209, var_get 127, and var_put 129.

dec_var does the same thing as inc_var, except does a decrement.

164b ⟨*Decrement variable* 164b⟩≡ (213)

```
dec_var:
    SUBROUTINE

    LDA     OPERAND0
    JSR     var_get
    SUBB    SCRATCH2, #$01
    JMP     inc_var_continue
```

Defines:
dec_var, used in chunks 174a and 180b.
Uses OPERAND0 209, SCRATCH2 209, SUBB 19a, and var_get 127.

15.1.1 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.

165a ⟨*Handle branch* 165a⟩≡ (212b) 165b ▷

```
branch:
    SUBROUTINE

    JSR     get_next_code_byte
    ORA     #$00
    BMI     .do_branch
    BPL     .no_branch     ; unconditional

negated_branch:
    JSR     get_next_code_byte
    ORA     #$00
    BPL     .do_branch
```

Defines:
  branch, used in chunks 155c, 159a, 181, 182, and 184b.
  negated_branch, used in chunks 156a, 159b, 181–85, and 192.
Uses get_next_code_byte 45.

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.

165b ⟨*Handle branch* 165a⟩+≡ (212b) ◁ 165a 166 ▷

```
.no_branch:
    AND     #$40
    BNE     .next
    JSR     get_next_code_byte

.next:
    JMP     do_instruction
```

Uses do_instruction 117 and get_next_code_byte 45.

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.

166 ⟨*Handle branch* 165a⟩+≡ (212b) ◁165b 167a▷

```
.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` 209 and `get_next_code_byte` 45.

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.

167a ⟨*Handle branch* 165a⟩+≡ (212b) ◁166 167b▷

```
  .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 209, instr_rfalse 187b, and instr_rtrue 188a.

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.

167b ⟨*Handle branch* 165a⟩+≡ (212b) ◁167a 167c▷

```
  branch_to_offset:
      SUBROUTINE

      SUBB      SCRATCH2, #$01
```

Defines:
  branch_to_offset, used in chunk 186a.
Uses SCRATCH2 209 and SUBB 19a.

Next, we store twice the high byte of SCRATCH2 into SCRATCH1.

167c ⟨*Handle branch* 165a⟩+≡ (212b) ◁167b 168▷

```
      LDA       SCRATCH2+1
      STA       SCRATCH1
      ASL
      LDA       #$00
      ROL
      STA       SCRATCH1+1
```

Uses SCRATCH1 209 and SCRATCH2 209.

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.

168 ⟨*Handle branch* 165a⟩+≡ (212b) ◁167c

```
    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 209, SCRATCH2 209, ZCODE_PAGE_VALID 209, Z_PC 209, and do_instruction 117.

15.2 Data movement instructions

15.2.1 load

`load` loads the variable in the operand into the variable in the next code byte.

169a ⟨*Instruction load* 169a⟩≡ (213)

```
  instr_load:
      SUBROUTINE

      LDA       OPERAND0
      JSR       var_get
      JMP       store_and_next
```

Defines:
  `instr_load`, used in chunk 114.

Uses `OPERAND0` 209, `store_and_next` 163a, and `var_get` 127.

15.2.2 loadw

`loadw` loads a word from the array at the address given `OPERAND0`, indexed by `OPERAND1`, into the variable in the next code byte.

169b ⟨*Instruction loadw* 169b⟩≡ (213)

```
  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 114.

Uses `ADDW` 18a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, `get_next_code_word` 52a, `load_address` 52b, and `store_and_next` 163a.

15.2.3 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.

170a ⟨*Instruction loadb* 170a⟩≡ (213)

```
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 114.

Uses `ADDW` 18a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, `get_next_code_byte2` 50, `load_address` 52b, and `store_and_next` 163a.

15.2.4 store

`store` stores `OPERAND1` into the variable in `OPERAND0`.

170b ⟨*Instruction store* 170b⟩≡ (213)

```
instr_store:
    SUBROUTINE

    MOVW    OPERAND1, SCRATCH2
    LDA     OPERAND0

stretch_var_put:
    JSR     var_put
    JMP     do_instruction
```

Defines:
`instr_store`, used in chunk 114.
`stretch_var_put`, used in chunk 173a.

Uses `MOVW` 14a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, `do_instruction` 117, and `var_put` 129.

15.2.5 storew

`storew` stores `OPERAND2` into the word array pointed to by z-address `OPERAND0` at the index `OPERAND1`.

171 ⟨*Instruction storew* 171⟩≡ (213)

```
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 114.

Uses `ADDW` 18a, `OPERAND0` 209, `OPERAND1` 209, `OPERAND2` 209, `SCRATCH2` 209, and `do_instruction` 117.

15.2.6 storeb

`storeb` stores the low byte of `OPERAND2` into the byte array pointed to by z-address `OPERAND0` at the index `OPERAND1`.

172a ⟨*Instruction storeb* 172a⟩≡ (213)
```
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 114.
Uses `ADDW` 18a, `OPERAND0` 209, `OPERAND1` 209, `OPERAND2` 209, `SCRATCH2` 209, and `do_instruction` 117.

15.3 Stack instructions

15.3.1 pop

`pop` pops the stack. This throws away the popped value.

172b ⟨*Instruction pop* 172b⟩≡ (213)
```
instr_pop:
    SUBROUTINE

    JSR     pop
    JMP     do_instruction
```
Defines:
`instr_pop`, used in chunk 114.
Uses `do_instruction` 117 and `pop` 44.

15.3.2 pull

`pull` pops the top value off the stack and puts it in the variable in `OPERAND0`.

173a ⟨*Instruction pull* 173a⟩≡ (213)
```
  instr_pull:
      SUBROUTINE

      JSR     pop
      LDA     OPERAND0
      JMP     stretch_var_put
```
Defines:
  `instr_pull`, used in chunk 114.
Uses `OPERAND0` 209, `pop` 44, and `stretch_var_put` 170b.

15.3.3 push

`push` pushes the value in `OPERAND0` onto the z-stack.

173b ⟨*Instruction push* 173b⟩≡ (213)
```
  instr_push:
      SUBROUTINE

      MOVW    OPERAND0, SCRATCH2
      JSR     push
      JMP     do_instruction
```
Defines:
  `instr_push`, used in chunk 114.
Uses `MOVW` 14a, `OPERAND0` 209, `SCRATCH2` 209, `do_instruction` 117, and `push` 43.

15.4 Decrements and increments

15.4.1 inc

`inc` increments the variable in the operand.

173c ⟨*Instruction inc* 173c⟩≡ (213)
```
  instr_inc:
      SUBROUTINE

      JSR     inc_var
      JMP     do_instruction
```
Defines:
  `instr_inc`, used in chunk 114.
Uses `do_instruction` 117 and `inc_var` 164a.

15.4.2 dec

`dec` decrements the variable in the operand.

174a ⟨*Instruction dec* 174a⟩≡ (213)

```
instr_dec:
    SUBROUTINE

    JSR     dec_var
    JMP     do_instruction
```

Defines:
  `instr_dec`, used in chunk 114.
Uses `dec_var` 164b and `do_instruction` 117.

15.5 Arithmetic instructions

15.5.1 add

`add` adds the first operand to the second operand and stores the result in the variable in the next code byte.

174b ⟨*Instruction add* 174b⟩≡ (213)

```
instr_add:
    SUBROUTINE

    ADDW    OPERAND0, OPERAND1, SCRATCH2
    JMP     store_and_next
```

Defines:
  `instr_add`, used in chunk 114.
Uses `ADDW` 18a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, and `store_and_next` 163a.

15.5.2 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.

175 ⟨*Instruction div* 175⟩≡ (213)

```
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 114.

Uses `MOVW` 14a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, `check_sign` 100b, and `divu16` 105.

15.5.3 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.

176 ⟨*Instruction mod* 176⟩≡ (213)

```
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 114.

Uses `MOVW` 14a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, `check_sign` 100b, `divu16` 105, and `store_and_next` 163a.

15.5.4 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.

177 ⟨*Instruction mul* 177⟩≡ (213)

```
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 114.

Uses `MOVW` 14a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, `check_sign` 100b, `mulu16` 102, `set_sign` 101, and `store_and_next` 163a.

15.5.5 random

`random` gets a random number between 1 and `OPERAND0`.

178a ⟨*Instruction random* 178a⟩≡ (213)

```
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 114.
Uses `INCW` 15b, `MOVW` 14a, `OPERAND0` 209, `SCRATCH1` 209, `SCRATCH2` 209, `divu16` 105, `get_random` 178b, and `store_and_next` 163a.

178b ⟨*Get random* 178b⟩≡ (213)

```
get_random:
    SUBROUTINE

    ROL     RANDOM_VAL+1
    MOVW    RANDOM_VAL, SCRATCH2
    RTS
```

Defines:
  `get_random`, used in chunk 178a.
Uses `MOVW` 14a and `SCRATCH2` 209.

15.5.6 sub

`sub` subtracts the first operand from the second operand and stores the result in the variable in the next code byte.

178c ⟨*Instruction sub* 178c⟩≡ (213)

```
instr_sub:
    SUBROUTINE

    SUBW    OPERAND0, OPERAND1, SCRATCH2
    JMP     store_and_next
```

Defines:
  `instr_sub`, used in chunk 114.
Uses `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, `SUBW` 20a, and `store_and_next` 163a.

15.6 Logical instructions

15.6.1 and

`and` bitwise-ands the first operand with the second operand and stores the result in the variable given by the next code byte.

179a ⟨*Instruction and* 179a⟩≡ (213)

```
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 114.
Uses `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, and `store_and_next` 163a.

15.6.2 not

`not` flips every bit in the variable in the operand and stores it in the variable in the next code byte.

179b ⟨*Instruction not* 179b⟩≡ (213)

```
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 114.
Uses `OPERAND0` 209, `SCRATCH2` 209, and `store_and_next` 163a.

15.6.3 or

`or` bitwise-ors the first operand with the second operand and stores the result in the variable given by the next code byte.

180a ⟨*Instruction or* 180a⟩≡ (213)

```
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 114.

Uses `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, and `store_and_next` 163a.

15.7 Conditional branch instructions

15.7.1 dec_chk

`dec_chk` decrements the variable in the first operand, and then jumps if it is less than the second operand.

180b ⟨*Instruction dec chk* 180b⟩≡ (213)

```
instr_dec_chk:
    SUBROUTINE

    JSR     dec_var
    MOVW    OPERAND1, SCRATCH1
    JMP     do_chk
```

Defines:
`instr_dec_chk`, used in chunk 114.

Uses `MOVW` 14a, `OPERAND1` 209, `SCRATCH1` 209, `dec_var` 164b, and `do_chk` 181a.

15.7.2 inc_chk

`inc_chk` increments the variable in the first operand, and then jumps if it is greater than the second operand.

181a ⟨*Instruction inc chk* 181a⟩≡ (213)

```
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 180b.
`instr_inc_chk`, used in chunk 114.
`stretch_to_branch`, used in chunks 183–85.
Uses `MOVW` 14a, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, `branch` 165a, `cmp16` 106b, `inc_var` 164a, and `negated_branch` 165a.

15.7.3 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`.

181b ⟨*Instruction je* 181b⟩≡ (213) 182a ▷

```
instr_je:
    SUBROUTINE

    LDX     OPERAND_COUNT
    DEX
    BNE     .check_second
    JSR     brk
```

Defines:
`instr_je`, used in chunk 114.
Uses `OPERAND_COUNT` 209 and `brk` 39b.

Next, we check against the second operand, and if it's equal, we branch, and if that was the last operand, we negative branch.

182a ⟨*Instruction je* 181b⟩+≡ (213) ◁181b 182b▷

```
  .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` 209, `OPERAND1` 209, and `branch` 165a.

Next we do the same with the third operand.

182b ⟨*Instruction je* 181b⟩+≡ (213) ◁182a 182c▷

```
      LDA       OPERAND0
      CMP       OPERAND0+4
      BNE       .check_next2
      LDA       OPERAND0+1
      CMP       OPERAND0+5
      BEQ       .branch

  .check_next2:
      DEX
      BEQ       .neg_branch
```

Uses `OPERAND0` 209 and `branch` 165a.

And again with the fourth operand.

182c ⟨*Instruction je* 181b⟩+≡ (213) ◁182b

```
      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` 209, `branch` 165a, and `negated_branch` 165a.

15.7.4 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.

183a ⟨*Instruction jg* 183a⟩≡ (213)
```
  instr_jg:
      SUBROUTINE

      MOVW      OPERAND0, SCRATCH1
      MOVW      OPERAND1, SCRATCH2
      JSR       cmp16
      BCC       stretch_to_branch
      JMP       negated_branch
```
Defines:
  `instr_jg`, used in chunk 114.
Uses `MOVW` 14a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, `cmp16` 106b, `negated_branch` 165a, and `stretch_to_branch` 181a.

15.7.5 jin

`jin` jumps if the first operand is a child object of the second operand.

183b ⟨*Instruction jin* 183b⟩≡ (213)
```
  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 114.
Uses `OBJECT_PARENT_OFFSET` 211a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH2` 209, `get_object_addr` 137, `negated_branch` 165a, and `stretch_to_branch` 181a.

15.7.6 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.

184a ⟨*Instruction jl* 184a⟩≡ (213)

```
instr_jl:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    MOVW     OPERAND1, SCRATCH1
    JSR      cmp16
    BCC      stretch_to_branch
    JMP      negated_branch
```

Defines:
`instr_jl`, used in chunk 114.

Uses `MOVW` 14a, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, `cmp16` 106b, `negated_branch` 165a, and `stretch_to_branch` 181a.

15.7.7 jz

`jz` jumps if its operand is `0`.

This also includes a "stretchy jump" for other instructions that need to branch.

184b ⟨*Instruction jz* 184b⟩≡ (213)

```
instr_jz:
    SUBROUTINE

    LDA      OPERAND0+1
    ORA      OPERAND0
    BEQ      take_branch
    JMP      negated_branch

take_branch:
    JMP      branch
```

Defines:
`instr_jz`, used in chunk 114.
`take_branch`, used in chunk 192.

Uses `OPERAND0` 209, `branch` 165a, and `negated_branch` 165a.

15.7.8 test

`test` jumps if all the bits in the first operand are set in the second operand.

185a ⟨*Instruction test* 185a⟩≡ (213)

```
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 114.

Uses `MOVB` 13b, `OPERAND0` 209, `OPERAND1` 209, `SCRATCH1` 209, `SCRATCH2` 209, `cmpu16` 106a, `negated_branch` 165a, and `stretch_to_branch` 181a.

15.7.9 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.

185b ⟨*Instruction test attr* 185b⟩≡ (213)

```
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 114.

Uses `SCRATCH1` 209, `SCRATCH3` 209, `attr_ptr_and_mask` 142, `negated_branch` 165a, and `stretch_to_branch` 181a.

15.8 Jump and subroutine instructions

15.8.1 call

call calls the routine at the given address.This instruction has been described in Call.

15.8.2 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.

186a ⟨*Instruction jump* 186a⟩≡ (213)

```
instr_jump:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    SUBB     SCRATCH2, #$01
    JMP      branch_to_offset
```

Defines:
instr_jump, used in chunk 114.
Uses MOVW 14a, OPERAND0 209, SCRATCH2 209, SUBB 19a, and branch_to_offset 167b.

15.8.3 print_ret

print_ret is the same as print, except that it prints a CRLF after the string, and then calls the rtrue instruction.

186b ⟨*Instruction print ret* 186b⟩≡ (213)

```
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 114.
Uses buffer_char 61 and instr_rtrue 188a.

15.8.4 ret

`ret` returns from a routine. The operand is the return value. This instruction has been described in Return.

15.8.5 ret_popped

`ret_popped` pops the stack and returns that value.

187a ⟨*Instruction ret popped* 187a⟩≡ (213)

```
instr_ret_popped:
    SUBROUTINE

    JSR     pop
    MOVW    SCRATCH2, OPERAND0
    JMP     instr_ret
```

Defines:
`instr_ret_popped`, used in chunk 114.
Uses `MOVW` 14a, `OPERAND0` 209, `SCRATCH2` 209, `instr_ret` 134, and `pop` 44.

15.8.6 rfalse

`rfalse` places `#$0000` into `OPERAND0`, and then calls the `ret` instruction.

187b ⟨*Instruction rfalse* 187b⟩≡ (213)

```
instr_rfalse:
    SUBROUTINE

    LDA     #$00
    JMP     ret_a
```

Defines:
`instr_rfalse`, used in chunks 114 and 167a.
Uses `ret_a` 188a.

15.8.7 rtrue

`rtrue` places `#$0001` into `OPERAND0`, and then calls the `ret` instruction.

188a ⟨*Instruction rtrue* 188a⟩≡ (213)

```
instr_rtrue:
    SUBROUTINE

    LDA      #$01
ret_a:
    STA      OPERAND0
    LDA      #$00
    STA      OPERAND0+1
    JMP      instr_ret
```

Defines:
`instr_rtrue`, used in chunks 114, 167a, and 186b.
`ret_a`, used in chunk 187b.
Uses `OPERAND0` 209 and `instr_ret` 134.

15.9 Print instructions

15.9.1 new_line

`new_line` prints `CRLF`.

188b ⟨*Instruction new line* 188b⟩≡ (213)

```
instr_new_line:
    SUBROUTINE

    LDA      #$0D
    JSR      buffer_char
    LDA      #$0A
    JSR      buffer_char
    JMP      do_instruction
```

Defines:
`instr_new_line`, used in chunk 114.
Uses `buffer_char` 61 and `do_instruction` 117.

15.9.2 print

print treats the following bytes of z-code as a z-encoded string, and prints it to the output.

189a ⟨*Instruction print* 189a⟩≡ (213)
```
instr_print:
    SUBROUTINE

    JSR     print_string_literal
    JMP     do_instruction
```
Defines:
instr_print, used in chunk 114.
Uses do_instruction 117.

15.9.3 print_addr

print_addr prints the z-encoded string at the address given by the operand.

189b ⟨*Instruction print addr* 189b⟩≡ (213)
```
instr_print_addr:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2
    JSR     load_address
    JMP     print_zstring_and_next
```
Defines:
instr_print_addr, used in chunk 114.
Uses MOVW 14a, OPERAND0 209, SCRATCH2 209, load_address 52b, and print_zstring_and_next 163b.

15.9.4 print_char

print_char prints the one-byte ASCII character in OPERAND0.

189c ⟨*Instruction print char* 189c⟩≡ (213)
```
instr_print_char:
    SUBROUTINE

    LDA     OPERAND0
    JSR     buffer_char
    JMP     do_instruction
```
Defines:
instr_print_char, used in chunk 114.
Uses OPERAND0 209, buffer_char 61, and do_instruction 117.

15.9.5 print_num

print_num prints the 16-bit signed value in OPERAND0 as a decimal number.

190a ⟨*Instruction print num* 190a⟩≡ (213)

```
instr_print_num:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    JSR      print_number
    JMP      do_instruction
```

Defines:
instr_print_num, used in chunk 114.
Uses MOVW 14a, OPERAND0 209, SCRATCH2 209, do_instruction 117, and print_number 108.

15.9.6 print_obj

print_obj prints the short name of the object in the operand.

190b ⟨*Instruction print obj* 190b⟩≡ (213)

```
instr_print_obj:
    SUBROUTINE

    LDA      OPERAND0
    JSR      print_obj_in_A
    JMP      do_instruction
```

Defines:
instr_print_obj, used in chunk 114.
Uses OPERAND0 209, do_instruction 117, and print_obj_in_A 141b.

15.9.7 print_paddr

print_paddr prints the z-encoded string at the packed address in the operand.

190c ⟨*Instruction print paddr* 190c⟩≡ (213)

```
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 114.
Uses MOVW 14a, OPERAND0 209, SCRATCH2 209, load_packed_address 53,
and print_zstring_and_next 163b.

15.10 Object instructions

15.10.1 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.

191 ⟨*Instruction clear attr* 191⟩≡ (213)

```
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 114.

Uses `SCRATCH1` 209, `SCRATCH2` 209, `SCRATCH3` 209, `attr_ptr_and_mask` 142, and `do_instruction` 117.

15.10.2 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`).

192 ⟨*Instruction get child* 192⟩≡ (213)

```
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 200.

Uses `OBJECT_CHILD_OFFSET` 211a, `OPERAND0` 209, `SCRATCH2` 209, `get_object_addr` 137, `negated_branch` 165a, `store_var` 128, and `take_branch` 184b.

15.10.3 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.

193 ⟨*Instruction get next prop* 193⟩≡ (213)

```
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 114.

Uses `OPERAND1` 209, `get_property_num` 145a, `get_property_ptr` 144, `next_property` 146, `store_A_and_next` 163a, and `store_zero_and_next` 163a.

15.10.4 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.

194 ⟨*Instruction get parent* 194⟩≡ (213)

```
instr_get_parent:
    SUBROUTINE

    LDA     OPERAND0
    JSR     get_object_addr
    LDY     #OBJECT_PARENT_OFFSET
    LDA     (SCRATCH2),Y
    STA     SCRATCH2
    LDA     #$00
    STA     SCRATCH2+1
    JMP     store_and_next
```

Defines:
`instr_get_parent`, used in chunk 114.

Uses `OBJECT_PARENT_OFFSET` 211a, `OPERAND0` 209, `SCRATCH2` 209, `get_object_addr` 137, and `store_and_next` 163a.

15.10.5 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.

195 ⟨*Instruction get prop* 195⟩≡ (213) 196 ▷

```
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 114.

Uses `OPERAND1` 209, `get_property_num` 145a, `get_property_ptr` 144, and `next_property` 146.

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.

196 ⟨*Instruction get prop* 195⟩+≡ (213) ◁195 197▷

```
.get_default:
    LDY      #HEADER_OBJECT_TABLE_ADDR+1
    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 211a, OPERAND1 209, SCRATCH1 209, SCRATCH2 209, and store_and_next 163a.

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.

197 ⟨*Instruction get prop* 195⟩+≡ (213) ◁196

```
.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 14a, SCRATCH1 209, SCRATCH2 209, brk 39b, get_property_len 145b, and store_and_next 163a.

15.10.6 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.

198 ⟨*Instruction get prop addr* 198⟩≡ (213)

```
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 114.

Uses ADDAC 16b, INCW 15b, OPERAND1 209, SCRATCH2 209, SUBW 20a, get_property_num 145a, get_property_ptr 144, next_property 146, store_and_next 163a, and store_zero_and_next 163a.

15.10.7 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.

199 ⟨*Instruction get prop len* 199⟩≡ (213)

```
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 114.

Uses `OPERAND0` 209, `SCRATCH2` 209, `get_property_len` 145b, and `store_A_and_next` 163a.

15.10.8 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`).

200 ⟨*Instruction get sibling* 200⟩≡ (213)

```
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 114.
Uses `OBJECT_SIBLING_OFFSET` 211a, `OPERAND0` 209, `get_object_addr` 137, and `push_and_check_obj` 192.

15.10.9 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.

201 ⟨*Instruction insert obj* 201⟩≡ (213)

```
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 114.

Uses OBJECT_CHILD_OFFSET 211a, OBJECT_PARENT_OFFSET 211a, OBJECT_SIBLING_OFFSET 211a, OPERAND0 209, OPERAND1 209, PSHW 14b, PULW 15a, SCRATCH2 209, do_instruction 117, get_object_addr 137, and remove_obj 139a.

15.10.10 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.

202 ⟨*Instruction put prop* 202⟩≡ (213)

```
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 114.

Uses OPERAND1 209, OPERAND2 209, SCRATCH2 209, brk 39b, do_instruction 117, get_property_len 145b, get_property_num 145a, get_property_ptr 144, and next_property 146.

15.10.11 remove_obj

`remove_obj` removes the object in the operand from the object tree.

203a ⟨*Instruction remove obj* 203a⟩≡ (213)

```
instr_remove_obj:
    SUBROUTINE

    JSR     remove_obj
    JMP     do_instruction
```

Defines:
`instr_remove_obj`, used in chunk 114.
Uses `do_instruction` 117 and `remove_obj` 139a.

15.10.12 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.

203b ⟨*Instruction set attr* 203b⟩≡ (213)

```
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 114.
Uses `SCRATCH1` 209, `SCRATCH2` 209, `SCRATCH3` 209, `attr_ptr_and_mask` 142, and `do_instruction` 117.

15.11 Other instructions

15.11.1 nop

`nop` does nothing.

204a ⟨*Instruction nop* 204a⟩≡ (213)

```
instr_nop:
    SUBROUTINE

    JMP     do_instruction
```

Defines:
`instr_nop`, used in chunk 114.
Uses `do_instruction` 117.

15.11.2 restart

`restart` restarts the game. This dumps the buffer, and then jumps back to `main`.

204b ⟨*Instruction restart* 204b⟩≡ (213)

```
instr_restart:
    SUBROUTINE

    JSR     dump_buffer_with_more
    JMP     main
```

Defines:
`instr_restart`, used in chunk 114.
Uses `dump_buffer_with_more` 59 and `main` 33a.

15.11.3 restore

`restore` restores the game. See the section Restoring the game state.

15.11.4 quit

`quit` quits the game by printing "– END OF SESSION –" and then spinlooping.

205 ⟨*Instruction quit* 205⟩≡ (213)

```
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 114.
Uses `SCRATCH2` 209, `STOW` 12, `dump_buffer_with_more` 59, and `print_ascii_string` 63b.

15.11.5 save

`save` saves the game. See the section Saving the game state.

15.11.6 sread

`sread` reads a line of input from the keyboard and parses it. See the section Lexical parsing.

Chapter 16

The entire program

206a ⟨*boot1.asm* 206a⟩≡

```
        PROCESSOR 6502

    ⟨Macros 12⟩
    ⟨defines 207b⟩

        ORG         $0800

    ⟨BOOT1 23a⟩
```

206b ⟨*boot2.asm* 206b⟩≡

```
        PROCESSOR 6502

    ⟨Macros 12⟩
    ⟨Apple ROM defines 208⟩
    ⟨RWTS defines 249⟩

    main    EQU     $0800

        ORG         $2300

    ⟨BOOT2 27⟩
```

Uses `main` 33a.

207a ⟨*main.asm* 207a⟩≡

```
        PROCESSOR 6502

    ⟨Macros 12⟩
    ⟨defines 207b⟩

        ORG     $0800

    ⟨routines 213⟩
```

207b ⟨*defines* 207b⟩≡ (206a 207a)

```
    ⟨Apple ROM defines 208⟩
    ⟨Program defines 209⟩
    ⟨Table offsets 211a⟩
    ⟨variable numbers 211b⟩
```

208 ⟨*Apple ROM defines* 208⟩≡ (206b 207b)

```
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 59, 73, and 149.
- CLREOL, used in chunks 59, 73, and 149.
- COUT, used in chunks 57 and 60b.
- COUT1, used in chunks 54, 56, and 60a.
- CSW, used in chunks 57 and 60b.
- CV, used in chunk 73.
- GETLN1, used in chunk 75.
- HOME, used in chunk 55a.
- INIT, used in chunks 25a and 28a.
- INVFLG, used in chunks 55b, 59, 73, and 149.
- IWMDATAPTR, used in chunks 23b and 24d.
- IWMSECTOR, used in chunk 24c.
- IWMSLTNDX, used in chunks 23–25.
- PROMPT, used in chunk 55b.
- RDKEY, used in chunks 59, 149, 151a, and 152.
- RDSECT_PTR, used in chunks 23c and 24d.
- SETKBD, used in chunks 25a and 28a.
- SETVID, used in chunks 25a, 28a, and 247c.
- VTAB, used in chunk 73.
- WNDBTM, used in chunks 55b and 59.
- WNDLFT, used in chunk 55b.
- WNDTOP, used in chunks 55, 59, and 75.
- WNDWDTH, used in chunks 55b and 60–63.

209 ⟨*Program defines* 209⟩≡ (207b)

```
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
HIGH_MEM_ADDR       EQU     $B8
Z_HEADER_ADDR       EQU     $BA     ; 2 bytes
NUM_IMAGE_PAGES     EQU     $BC
NUM_PAGE_TABLE_ENTRIES EQU  $BD
FIRST_Z_PAGE        EQU     $BE
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:

BUFF_AREA, used in chunks 56, 57, 61–63, 75, 112, 113, 153b, 154a, 156c, and 157c.
BUFF_END, used in chunks 56, 57, 60–63, and 75.
BUFF_LINE_LEN, used in chunks 62b and 63a.
CURR_DISK_BUFF_ADDR, never used.
CURR_LINE, used in chunks 55a, 59, and 75.
CURR_OPCODE, used in chunks 117, 120–22, and 124.
DEBUG_JUMP, used in chunks 28a, 116, and 249.
FIRST_Z_PAGE, used in chunks 35b and 49.
FRAME_STACK_COUNT, used in chunks 131a and 133–35.
FRAME_Z_SP, used in chunks 131a and 133–35.
GLOBAL_ZVARS_ADDR, used in chunks 40, 126, 128, and 135b.
HIGH_MEM_ADDR, used in chunks 41, 47, and 50.
LAST_Z_PAGE, used in chunks 35b, 41, 48, and 49.
LOCAL_ZVARS, used in chunks 126, 128, 132, 133a, 154b, and 157b.
LOCKED_ALPHABET, used in chunks 64, 66, 68, 69, 85, 86b, 88a, and 90.
NEXT_PAGE_TABLE, used in chunks 34d, 35a, 41, and 49.
NUM_IMAGE_PAGES, used in chunks 37, 38a, 41, 46, and 50.
OPERAND0, used in chunks 75, 77, 79b, 80a, 83, 119d, 121a, 123, 130, 131b, 136, 139, 140, 142, 144, 164, 169–80, 182–90, 192, 194, and 199–201.
OPERAND1, used in chunks 77–79, 81, 82, 121b, 133a, 142, 169–72, 174–85, 193, 195, 196, 198, 201, and 202.
OPERAND2, used in chunks 171, 172a, and 202.
OPERAND3, never used.
OPERAND_COUNT, used in chunks 117, 119d, 122a, 123, 133a, and 181b.
PAGE_H_TABLE, used in chunks 34d, 35a, 47, 48, and 50.
PAGE_L_TABLE, used in chunks 34d, 35a, 47, 48, and 50.
PAGE_TABLE_INDEX, used in chunks 46, 47, and 50.
PAGE_TABLE_INDEX2, used in chunks 47 and 50.
PREV_PAGE_TABLE, used in chunks 34d, 35a, and 49.
PRINTER_CSW, used in chunks 34a, 57, and 60b.
RWTS, used in chunks 27, 111, 236, and 242.
SCRATCH1, used in chunks 36c, 38a, 47, 50, 82, 85–88, 90, 91, 93, 95–98, 100b, 102, 105, 106, 108, 111–13, 116, 126, 128, 133a, 135, 140–44, 151b, 167c, 168, 175–78, 180b, 181a, 183–85, 191, 196, 197, and 203b.
SCRATCH2, used in chunks 36c, 38a, 42–44, 46–50, 52–54, 59, 63b, 65, 69, 73, 84–87, 94–100, 102, 105, 106, 108, 111–13, 116, 118–26, 128–33, 135–45, 147, 149, 151–55, 157, 158, 163, 164, 166–81, 183–87, 189–92, 194, 196–99, 201–3, and 205.
SCRATCH3, used in chunks 63b, 67a, 68, 70–72, 77–83, 85, 86, 88c, 90–93, 95–97, 102, 105, 108, 133a, 135, 143a, 155b, 158b, 185b, 191, and 203b.
SECTORS_PER_TRACK, used in chunks 28a, 111, and 249.
SHIFT_ALPHABET, used in chunks 64, 66, 68, and 69.
STACK_COUNT, used in chunks 34c, 43, 44, 133b, 134, 154b, and 157b.
TMP_Z_PC, used in chunk 117.
ZCHARS_H, used in chunks 65 and 69.
ZCHARS_L, used in chunks 65 and 69.
ZCHAR_SCRATCH1, used in chunks 34c, 79, 80, 86a, and 87b.
ZCHAR_SCRATCH2, used in chunks 85, 88–91, 93, 96a, and 97.
ZCODE_PAGE_ADDR, used in chunks 45, 46, and 72b.
ZCODE_PAGE_ADDR2, used in chunks 50 and 72b.

ZCODE_PAGE_VALID, used in chunks 34b, 45, 46, 50, 72b, 131a, 136, 157b, and 168.
ZCODE_PAGE_VALID2, used in chunks 34b, 47, 50, 53, 69, and 72b.
ZDECOMPRESS_STATE, used in chunks 65, 66, and 69.
Z_ABBREV_TABLE, used in chunks 40 and 69.
Z_PC, used in chunks 39c, 45–47, 72b, 117, 126, 131, 135d, 153c, 157b, and 168.
Z_PC2_H, used in chunks 50, 52b, 53, 69, and 72b.
Z_PC2_HH, used in chunks 50, 52b, 53, 69, and 72b.
Z_PC2_L, used in chunks 50, 52b, 53, 69, and 72b.
Z_SP, used in chunks 34c, 43, 44, 133b, and 134.

211a ⟨*Table offsets* 211a⟩≡ (207b)

```
HEADER_VERSION          EQU     $00
HEADER_FLAGS1           EQU     $01
HEADER_HIMEM_BASE       EQU     $04
HEADER_INITIAL_ZPC      EQU     $06
HEADER_DICT_ADDR        EQU     $08
HEADER_OBJECT_TABLE_ADDR EQU    $0A
HEADER_GLOBALVARS_ADDR  EQU     $0C
HEADER_STATIC_MEM_BASE  EQU     $0E
HEADER_FLAGS2           EQU     $10
HEADER_ABBREVS_ADDR     EQU     $18
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 138a.
HEADER_DICT_ADDR, used in chunk 94.
HEADER_FLAGS2, used in chunks 58, 60b, 75, 157a, and 158c.
HEADER_OBJECT_TABLE_ADDR, used in chunks 138b and 196.
HEADER_STATIC_MEM_BASE, used in chunks 155b and 158b.
OBJECT_CHILD_OFFSET, used in chunks 139c, 140b, 192, and 201.
OBJECT_PARENT_OFFSET, used in chunks 139a, 140c, 183b, 194, and 201.
OBJECT_PROPS_OFFSET, used in chunks 141b and 144.
OBJECT_SIBLING_OFFSET, used in chunks 140, 200, and 201.

211b ⟨*variable numbers* 211b⟩≡ (207b)

```
VAR_CURR_ROOM       EQU     $10
VAR_SCORE           EQU     $11
VAR_MAX_SCORE       EQU     $12
```

Defines:
VAR_CURR_ROOM, used in chunk 73.
VAR_MAX_SCORE, used in chunk 73.
VAR_SCORE, used in chunk 73.

212a ⟨*Internal error string* 212a⟩≡ (213)

```
sInternalError:
      DC        "ZORK INTERNAL ERROR!"
```

Defines:
`sInternalError`, never used.

212b ⟨*Instruction execution routines* 212b⟩≡ (213)

```
⟨Instruction tables 114⟩

⟨Do instruction 117⟩
⟨Execute instruction 116⟩
⟨Handle 0op instructions 118b⟩
⟨Handle 1op instructions 119a⟩
⟨Handle 2op instructions 121a⟩
⟨Get const byte 125a⟩
⟨Get const word 125b⟩
⟨Get var content in A 127⟩
⟨Store to var A 129⟩
⟨Get var content 126⟩
⟨Store and go to next instruction 163a⟩
⟨Store var 128⟩
⟨Handle branch 165a⟩
```

212c ⟨*Disk routines* 212c⟩≡ (213)

```
⟨iob struct 110⟩
⟨Do RWTS on sector 111⟩
⟨Reading sectors 112⟩
⟨Writing sectors 113⟩
⟨Do reset window 36b⟩
⟨Print ASCII string 63b⟩
⟨Save diskette strings 148b⟩
⟨Insert save diskette 147⟩
⟨Get prompted number from user 149⟩
⟨Reinsert game diskette 152⟩
⟨Instruction save 153a⟩
⟨Copy data to buff 154a⟩
⟨Instruction restore 156b⟩
⟨Copy data from buff 157c⟩
```

213 ⟨*routines* 213⟩≡ (207a)

⟨*main* 33a⟩

⟨*Instruction execution routines* 212b⟩
⟨*Instruction rtrue* 188a⟩
⟨*Instruction rfalse* 187b⟩
⟨*Instruction print* 189a⟩
⟨*Printing a string literal* 72b⟩
⟨*Instruction print ret* 186b⟩
⟨*Instruction nop* 204a⟩
⟨*Instruction ret popped* 187a⟩
⟨*Instruction pop* 172b⟩
⟨*Instruction new line* 188b⟩
⟨*Instruction jz* 184b⟩
⟨*Instruction get sibling* 200⟩
⟨*Instruction get child* 192⟩
⟨*Instruction get parent* 194⟩
⟨*Instruction get prop len* 199⟩
⟨*Instruction inc* 173c⟩
⟨*Instruction dec* 174a⟩
⟨*Increment variable* 164a⟩
⟨*Decrement variable* 164b⟩
⟨*Instruction print addr* 189b⟩
⟨*Instruction illegal opcode* 162⟩
⟨*Instruction remove obj* 203a⟩
⟨*Remove object* 139a⟩
⟨*Instruction print obj* 190b⟩
⟨*Print object in A* 141b⟩
⟨*Instruction ret* 134⟩
⟨*Instruction jump* 186a⟩
⟨*Instruction print paddr* 190c⟩
⟨*Print zstring and go to next instruction* 163b⟩
⟨*Instruction load* 169a⟩
⟨*Instruction not* 179b⟩
⟨*Instruction jl* 184a⟩
⟨*Instruction jg* 183a⟩
⟨*Instruction dec chk* 180b⟩
⟨*Instruction inc chk* 181a⟩
⟨*Instruction jin* 183b⟩
⟨*Instruction test* 185a⟩
⟨*Instruction or* 180a⟩
⟨*Instruction and* 179a⟩
⟨*Instruction test attr* 185b⟩
⟨*Instruction set attr* 203b⟩
⟨*Instruction clear attr* 191⟩
⟨*Instruction store* 170b⟩
⟨*Instruction insert obj* 201⟩
⟨*Instruction loadw* 169b⟩
⟨*Instruction loadb* 170a⟩
⟨*Instruction get prop* 195⟩

⟨*Printing a CRLF* 71c⟩
⟨*Shifting alphabets* 68⟩
⟨*Printing an abbreviation* 69⟩
⟨*A mod 3* 107⟩
⟨*A2 table* 71a⟩
⟨*Get alphabet* 64⟩
⟨*Get next zchar* 65⟩
⟨*ASCII to Zchar* 85⟩
⟨*Search nonalpha table* 92b⟩
⟨*Get alphabet for char* 87a⟩
⟨*Z compress* 89⟩
⟨*Instruction restart* 204b⟩
⟨*Locate last RAM page* 42a⟩
⟨*Buffer a character* 61⟩
⟨*Dump buffer line* 58⟩
⟨*Dump buffer to printer* 57⟩
⟨*Dump buffer to screen* 56⟩
⟨*Dump buffer with more* 59⟩
⟨*Home* 55a⟩
⟨*Print status line* 73⟩
⟨*Output string to console* 54⟩
⟨*Read line* 75⟩
⟨*Reset window* 55b⟩
⟨*Disk routines* 212c⟩
⟨*Instruction quit* 205⟩
⟨*Internal error string* 212a⟩
⟨*brk* 39b⟩
⟨*Get random* 178b⟩

```
    HEX     00 00 00 00 00 00 00 00
    HEX     00 FC 19 00 00
```

Chapter 17

Defined Chunks

Chapter 18

Appendix: RWTS

Part of DOS within BOOT2, and presented without comment. Commented source code can be seen at cmosher01's annotated Apple II source repository.

221 ⟨*RWTS Prenibble routine* 221⟩≡ (250)

```
PRENIBBLE:
    ; Converts 256 bytes of data to 342 6-bit nibbles.
    SUBROUTINE

    LDX      #$00
    LDY      #$02

.loop1:
    DEY
    LDA      (PTR2BUF),Y
    LSR
    ROL      SECONDARY_BUFF,X
    LSR
    ROL      SECONDARY_BUFF,X
    STA      PRIMARY_BUFF,Y
    INX
    CPX      #$56
    BCC      .loop1
    LDX      #$00
    TYA
    BNE      .loop1
    LDX      #$55

.loop2:
    LDA      SECONDARY_BUFF,X
    AND      #$3F
    STA      SECONDARY_BUFF,X
    DEX
```

```
      BPL       .loop2
      RTS
```

Defines:
  PRENIBBLE, used in chunk 236.

Uses PRIMARY_BUFF 233a and SECONDARY_BUFF 233b.

223 ⟨*RWTS Write routine* 223⟩≡ (250)

```
WRITE:
    ; Writes a sector to disk.
    SUBROUTINE

    SEC
    STX         RWTS_SCRATCH2
    STX         SLOTPG6
    LDA         Q6H,X
    LDA         Q7L,X
    BMI         .protected
    LDA         SECONDARY_BUFF
    STA         RWTS_SCRATCH
    LDA         #$FF
    STA         Q7H,X
    ORA         Q6L,X
    PHA
    PLA
    NOP
    LDY         #$04

.write_4_ff:
    PHA
    PLA
    JSR         WRITE2
    DEY
    BNE         .write_4_ff

    LDA         #$D5
    JSR         WRITE1
    LDA         #$AA
    JSR         WRITE1
    LDA         #$AD
    JSR         WRITE1
    TYA
    LDY         #$56
    BNE         .do_eor

.get_nibble:
    LDA         SECONDARY_BUFF,Y

.do_eor:
    EOR         SECONDARY_BUFF-1,Y
    TAX
    LDA         WRITE_XLAT_TABLE,X
    LDX         RWTS_SCRATCH2
    STA         Q6H,X
    LDA         Q6L,X
    DEY
    BNE         .get_nibble
```

```
    LDA     RWTS_SCRATCH
    NOP

.second_eor:
    EOR     PRIMARY_BUFF,Y
    TAX
    LDA     WRITE_XLAT_TABLE,X
    LDX     SLOTPG6
    STA     Q6H,X
    LDA     Q6L,X
    LDA     PRIMARY_BUFF,Y
    INY
    BNE     .second_eor

    TAX
    LDA     WRITE_XLAT_TABLE,X
    LDX     RWTS_SCRATCH2
    JSR     WRITE3
    LDA     #$DE
    JSR     WRITE1
    LDA     #$AA
    JSR     WRITE1
    LDA     #$EB
    JSR     WRITE1
    LDA     #$FF
    JSR     WRITE1
    LDA     Q7L,X

.protected:
    LDA     Q6L,X
    RTS
```

Defines:
WRITE, used in chunks 236 and 245.
Uses PRIMARY_BUFF 233a, SECONDARY_BUFF 233b, WRITE1 225a, WRITE2 225a, WRITE3 225a, and WRITE_XLAT_TABLE 232b.

225a ⟨*RWTS Write bytes* 225a⟩≡ (250)

```
WRITE1:
    SUBROUTINE

    CLC

WRITE2:
    SUBROUTINE

    PHA
    PLA

WRITE3:
    SUBROUTINE

    STA     Q6H,X
    ORA     Q6L,X
    RTS
```

Defines:
`WRITE1`, used in chunk 223.
`WRITE2`, used in chunk 223.
`WRITE3`, used in chunk 223.

225b ⟨*RWTS Postnibble routine* 225b⟩≡ (250)

```
POSTNIBBLE:
    ; Converts nibbled data to regular data in PTR2BUF.
    SUBROUTINE

    LDY     #$00

.loop:
    LDX     #$56

.loop2:
    DEX
    BMI     .loop
    LDA     PRIMARY_BUFF,Y
    LSR     SECONDARY_BUFF,X
    ROL
    LSR     SECONDARY_BUFF,X
    ROL
    STA     (PTR2BUF),Y
    INY
    CPY     RWTS_SCRATCH
    BNE     .loop2
    RTS
```

Defines:
`POSTNIBBLE`, used in chunk 236.
Uses `PRIMARY_BUFF` 233a and `SECONDARY_BUFF` 233b.

226 ⟨*RWTS Read routine* 226⟩≡ (250)

```
READ:
    ; Reads a sector from disk.
    SUBROUTINE

    LDY      #$20

.await_prologue:
    DEY
    BEQ      read_error

.await_prologue_d5:
    LDA      Q6L,X
    BPL      .await_prologue_d5

.check_for_d5:
    EOR      #$D5
    BNE      .await_prologue
    NOP

.await_prologue_aa:
    LDA      Q6L,X
    BPL      .await_prologue_aa
    CMP      #$AA
    BNE      .check_for_d5
    LDY      #$56

.await_prologue_ad:
    LDA      Q6L,X
    BPL      .await_prologue_ad
    CMP      #$AD
    BNE      .check_for_d5
    LDA      #$00

.loop:
    DEY
    STY      RWTS_SCRATCH

.await_byte1:
    LDY      Q6L,X
    BPL      .await_byte1
    EOR      ARM_MOVE_DELAY,Y
    LDY      RWTS_SCRATCH
    STA      SECONDARY_BUFF,Y
    BNE      .loop

.save_index:
    STY      RWTS_SCRATCH

.await_byte2:
```

```
      LDY     Q6L,X
      BPL     .await_byte2
      EOR     ARM_MOVE_DELAY,Y
      LDY     RWTS_SCRATCH
      STA     PRIMARY_BUFF,Y
      INY
      BNE     .save_index

  .read_checksum:
      LDY     Q6L,X
      BPL     .read_checksum
      CMP     ARM_MOVE_DELAY,Y
      BNE     read_error

  .await_epilogue_de:
      LDA     Q6L,X
      BPL     .await_epilogue_de
      CMP     #$DE
      BNE     read_error
      NOP

  .await_epilogue_aa:
      LDA     Q6L,X
      BPL     .await_epilogue_aa
      CMP     #$AA
      BEQ     good_read

  read_error:
      SEC
      RTS
```

Defines:
  READ, used in chunks 236, 243, and 245.
  read_error, used in chunk 228.
Uses ARM_MOVE_DELAY 231, PRIMARY_BUFF 233a, SECONDARY_BUFF 233b, and good_read 228.

228 ⟨*RWTS Read address 228*⟩≡ (250)

```
READ_ADDR:
    ; Reads an address header from disk.
    SUBROUTINE

    LDY      #$FC
    STY      RWTS_SCRATCH

.await_prologue:
    INY
    BNE      .await_prologue_d5
    INC      RWTS_SCRATCH
    BEQ      read_error

.await_prologue_d5:
    LDA      Q6L,X
    BPL      .await_prologue_d5

.check_for_d5:
    CMP      #$D5
    BNE      .await_prologue
    NOP

.await_prologue_aa:
    LDA      Q6L,X
    BPL      .await_prologue_aa
    CMP      #$AA
    BNE      .check_for_d5
    LDY      #$03

.await_prologue_96:
    LDA      Q6L,X
    BPL      .await_prologue_96
    CMP      #$96
    BNE      .check_for_d5
    LDA      #$00

.calc_checksum:
    STA      RWTS_SCRATCH2

.get_header:
    LDA      Q6L,X
    BPL      .get_header
    ROL
    STA      RWTS_SCRATCH

.read_header:
    LDA      Q6L,X
    BPL      .read_header
    AND      RWTS_SCRATCH
```

```
    STA     CKSUM_ON_DISK,Y
    EOR     RWTS_SCRATCH2
    DEY
    BPL     .calc_checksum
    TAY
    BNE     read_error

.await_epilogue_de:
    LDA     Q6L,X
    BPL     .await_epilogue_de
    CMP     #$DE
    BNE     read_error
    NOP

.await_epilogue_aa:
    LDA     Q6L,X
    BPL     .await_epilogue_aa
    CMP     #$AA
    BNE     read_error

good_read:
    CLC
    RTS
```

Defines:
`READ_ADDR`, used in chunks 236, 243, and 245.
`good_read`, used in chunks 47, 50, and 226.
Uses `read_error` 226.

230 ⟨*RWTS Seek absolute* 230⟩≡ (250)

```
SEEKABS:
    ; Moves disk arm to a given half-track.
    SUBROUTINE

    STX     SLOT16
    STA     DEST_TRACK
    CMP     CURR_TRACK
    BEQ     entry_off_end
    LDA     #$00
    STA     RWTS_SCRATCH

.save_curr_track:
    LDA     CURR_TRACK
    STA     RWTS_SCRATCH2
    SEC
    SBC     DEST_TRACK
    BEQ     .at_destination
    BCS     .move_down
    EOR     #$FF
    INC     CURR_TRACK
    BCC     .check_delay_index

.move_down:
    ADC     #$FE
    DEC     CURR_TRACK

.check_delay_index:
    CMP     RWTS_SCRATCH
    BCC     .check_within_steps
    LDA     RWTS_SCRATCH

.check_within_steps:
    CMP     #$0C
    BCS     .turn_on
    TAY

.turn_on:
    SEC
    JSR     ON_OR_OFF
    LDA     ON_TABLE,Y
    JSR     ARM_MOVE_DELAY
    LDA     RWTS_SCRATCH2
    CLC
    JSR     ENTRY_OFF
    LDA     OFF_TABLE,Y
    JSR     ARM_MOVE_DELAY
    INC     RWTS_SCRATCH
    BNE     .save_curr_track
```

```
.at_destination:
    JSR      ARM_MOVE_DELAY
    CLC

ON_OR_OFF:
    LDA      CURR_TRACK

ENTRY_OFF:
    AND      #$03
    ROL
    ORA      SLOT16
    TAX
    LDA      PHASEOFF,X
    LDX      SLOT16

entry_off_end:
    RTS

garbage:
    HEX      AA A0 A0
```

Defines:
ENTRY_OFF, never used.
ON_OR_OFF, never used.
SEEKABS, used in chunk 241a.
entry_off_end, never used.
Uses ARM_MOVE_DELAY 231, OFF_TABLE 232a, and ON_TABLE 232a.

231 ⟨*RWTS Arm move delay* 231⟩≡ (250)

```
ARM_MOVE_DELAY:
    ; Delays during arm movement.
    SUBROUTINE

    LDX      #$11

.delay1:
    DEX
    BNE      .delay1
    INC      MOTOR_TIME
    BNE      .delay2
    INC      MOTOR_TIME+1

.delay2:
    SEC
    SBC      #$01
    BNE      ARM_MOVE_DELAY
    RTS
```

Defines:
ARM_MOVE_DELAY, used in chunks 226, 230, and 236.

232a ⟨*RWTS Arm move delay tables* 232a⟩≡ (250)

```
ON_TABLE:
      HEX       01 30 28 24 20 1E 1D 1C 1C 1C 1C 1C

OFF_TABLE:
      HEX       70 2C 26 22 1F 1E 1D 1C 1C 1C 1C 1C
```

Defines:
OFF_TABLE, used in chunk 230.
ON_TABLE, used in chunk 230.

232b ⟨*RWTS Write translate table* 232b⟩≡ (250)

```
WRITE_XLAT_TABLE:
      HEX       96 97 9A 9B 9D 9E 9F A6 A7 AB AC AD AE AF B2 B3
      HEX       B4 B5 B6 B7 B9 BA BB BC BD BE BF CB CD CE CF D3
      HEX       D6 D7 D9 DA DB DC DD DE DF E5 E6 E7 E9 EA EB EC
      HEX       ED EE EF F2 F3 F4 F5 F6 F7 F9 FA FB FC FD FE FF
```

Defines:
WRITE_XLAT_TABLE, used in chunk 223.

232c ⟨*RWTS Unused area* 232c⟩≡ (250)

```
      HEX       B3 B3 A0 E0 B3 C3 C5 B3 A0 E0 B3 C3 C5 B3 A0 E0
      HEX       B3 B3 C5 AA A0 82 B3 B3 C5 AA A0 82 C5 B3 B3 AA
      HEX       88 82 C5 B3 B3 AA 88 82 C5 C4 B3 B0 88
```

232d ⟨*RWTS Read translate table* 232d⟩≡ (250)

```
READ_XLAT_TABLE:
      HEX       00 01 98 99 02 03 9C 04 05 06 A0 A1 A2 A3 A4 A5
      HEX       07 08 A8 A9 AA 09 0A 0B 0C 0D B0 B1 0E 0F 10 11
      HEX       12 13 B8 14 15 16 17 18 19 1A C0 C1 C2 C3 C4 C5
      HEX       C6 C7 C8 C9 CA 1B CC 1C 1D 1E D0 D1 D2 1F D4 D5
      HEX       20 21 D8 22 23 24 25 26 27 28 E0 E1 E2 E3 E4 29
      HEX       2A 2B E8 2C 2D 2E 2F 30 31 32 F0 F1 33 34 35 36
      HEX       37 38 F8 39 3A 3B 3C 3D 3E 3F
```

Defines:
READ_XLAT_TABLE, never used.

233a ⟨*RWTS Primary buffer* 233a⟩≡ (250)

```
PRIMARY_BUFF:
    ; Initially contains this garbage.
    HEX     00 38 11 0A 08 20 20 0E 18 06 02 31 02 09 08 27
    HEX     22 00 12 0A 0A 04 00 00 03 2A 00 04 00 00 22 08
    HEX     10 28 12 02 00 02 08 11 0A 08 02 28 11 01 39 22
    HEX     31 01 05 18 20 28 02 10 06 02 09 02 05 2C 10 00
    HEX     08 2E 00 05 02 28 18 02 30 23 02 20 32 04 11 02
    HEX     14 02 08 09 12 20 0E 2F 23 30 2F 23 30 0C 17 2A
    HEX     3F 27 23 30 37 23 30 12 1A 08 30 0F 08 30 0F 27
    HEX     23 30 37 23 30 3A 22 34 3C 2A 35 08 35 0F 2A 2A
    HEX     08 35 0F 2A 25 08 35 0F 29 10 08 31 0F 29 11 08
    HEX     31 0F 29 0F 08 31 0F 29 10 11 11 11 0F 12 12 01
    HEX     0F 27 23 30 2F 23 30 1A 02 2A 08 35 0F 2A 37 08
    HEX     35 0F 2A 2A 08 35 0F 2A 3A 08 35 0F 06 2F 23 30
    HEX     2F 23 30 18 12 12 01 0F 27 23 30 37 23 30 1A 3A
    HEX     3A 3A 02 2A 3A 3A 12 1A 27 23 30 37 23 30 18 22
    HEX     29 3A 24 28 25 22 25 3A 24 28 25 22 25 24 24 32
    HEX     25 34 25 24 24 32 25 34 25 24 28 32 28 29 21 29
```

Defines:
PRIMARY_BUFF, used in chunks 221, 223, 225b, 226, and 243.

233b ⟨*RWTS Secondary buffer* 233b⟩≡ (250)

```
SECONDARY_BUFF:
    ; Initially contains this garbage.
    HEX     00 E1 45 28 21 82 80 38 62 19 0B C5 0B 24 21 9C
    HEX     88 00 48 28 2B 10 00 03 0C A9 01 10 01 00 88 22
    HEX     40 A0 48 09 01 08 21 44 29 22 08 A0 45 06 E4 8A
    HEX     C4 06 16 60 80 A0 09 40 18 0A 24 0A 16 B0 43 00
    HEX     20 BB 00 14 08 A0 60 0A C0 8F 0A 83 CA 11 44 08
    HEX     51 0A 20 26 4A 80
```

Defines:
SECONDARY_BUFF, used in chunks 221, 223, 225b, and 226.

234 ⟨*RWTS Write address header 234*⟩≡ (250)

```
WRITE_ADDR_HDR:
    SUBROUTINE

    SEC
    LDA     Q6H,X
    LDA     Q7L,X
    BMI     .set_read_mode
    LDA     #$FF
    STA     Q7H,X
    CMP     Q6L,X
    PHA
    PLA

.write_sync:
    JSR     WRITE_ADDR_RET
    JSR     WRITE_ADDR_RET
    STA     Q6H,X
    CMP     Q6L,X
    NOP
    DEY
    BNE     .write_sync
    LDA     #$D5
    JSR     WRITE_BYTE3
    LDA     #$AA
    JSR     WRITE_BYTE3
    LDA     #$96
    JSR     WRITE_BYTE3
    LDA     FORMAT_VOLUME
    JSR     WRITE_DOUBLE_BYTE
    LDA     FORMAT_TRACK
    JSR     WRITE_DOUBLE_BYTE
    LDA     FORMAT_SECTOR
    JSR     WRITE_DOUBLE_BYTE
    LDA     FORMAT_VOLUME
    EOR     FORMAT_TRACK
    EOR     FORMAT_SECTOR
    PHA
    LSR
    ORA     PTR2BUF
    STA     Q6H,X
    LDA     Q6L,X
    PLA
    ORA     #$AA
    JSR     WRITE_BYTE2
    LDA     #$DE
    JSR     WRITE_BYTE3
    LDA     #$AA
    JSR     WRITE_BYTE3
    LDA     #$EB
```

```
      JSR      WRITE_BYTE3
      CLC

  .set_read_mode:
      LDA      Q7L,X
      LDA      Q6L,X

  WRITE_ADDR_RET:
      RTS
```

Defines:
WRITE_ADDR_HDR, used in chunk 245.
Uses WRITE_BYTE2 235a, WRITE_BYTE3 235a, and WRITE_DOUBLE_BYTE 235a.

235a ⟨*RWTS Write address header bytes* 235a⟩≡ (250)

```
  WRITE_DOUBLE_BYTE:
      PHA
      LSR
      ORA      PTR2BUF
      STA      Q6H,X
      CMP      Q6L,X
      PLA
      NOP
      NOP
      NOP
      ORA      #$AA

  WRITE_BYTE2:
      NOP

  WRITE_BYTE3:
      NOP
      PHA
      PLA
      STA      Q6H,X
      CMP      Q6L,X
      RTS
```

Defines:
WRITE_BYTE2, used in chunk 234.
WRITE_BYTE3, used in chunk 234.
WRITE_DOUBLE_BYTE, used in chunk 234.

235b ⟨*RWTS Unused area 2* 235b⟩≡ (250)

```
      HEX      88 A5 E8 91 A0 94 88 96
      HEX      E8 91 A0 94 88 96 91 91
      HEX      C8 94 D0 96 91 91 C8 94
      HEX      D0 96 91 A3 C8 A0 A5 85
      HEX      A4
```

236 ⟨*RWTS Entry point* 236⟩≡ (250)

```
RWTS_entry:
    ; RWTS entry point.
    SUBROUTINE

    STY      PTR2IOB
    STA      PTR2IOB+1
    LDY      #$02
    STY      RECALIBCNT
    LDY      #$04
    STY      RESEEKCNT
    LDY      #$01
    LDA      (PTR2IOB),Y
    TAX
    LDY      #$0F
    CMP      (PTR2IOB),Y
    BEQ      .sameslot
    TXA
    PHA
    LDA      (PTR2IOB),Y
    TAX
    PLA
    PHA
    STA      (PTR2IOB),Y
    LDA      Q7L,X
.ck_spin:
    LDY      #$08
    LDA      Q6L,X
.check_change:
    CMP      Q6L,X
    BNE      .ck_spin
    DEY
    BNE      .check_change
    PLA
    TAX
.sameslot:
    LDA      Q7L,X
    LDA      Q6L,X
    LDY      #$08
.strobe_again:
    LDA      Q6L,X
    PHA
    PLA
    PHA
    PLA
    STX      SLOTPG5
    CMP      Q6L,X
    BNE      .done_test
    DEY
    BNE      .strobe_again
```

```
.done_test:
    PHP
    LDA     MOTORON,X
    LDY     #$06
.move_ptrs:
    LDA     (PTR2IOB),Y
    STA     PTR2DCT-6,Y
    INY
    CPY     #$0A
    BNE     .move_ptrs
    LDY     #$03
    LDA     (PTR2DCT),Y
    STA     MOTOR_TIME+1
    LDY     #$02
    LDA     (PTR2IOB),Y
    LDY     #$10
    CMP     (PTR2IOB),Y
    BEQ     .save_drive
    STA     (PTR2IOB),Y
    PLP
    LDY     #$00
    PHP
.save_drive:
    ROR
    BCC     .use_drive2
    LDA     DRV0EN,X
    BCS     .use_drive1
.use_drive2:
    LDA     DRV1EN,X
.use_drive1:
    ROR     ZPAGE_DRIVE
    PLP
    PHP
    BNE     .was_on
    LDY     #$07
.wait_for_motor:
    JSR     ARM_MOVE_DELAY
    DEY
    BNE     .wait_for_motor
    LDX     SLOTPG5
.was_on:
    LDY     #$04
    LDA     (PTR2IOB),Y
    JSR     rwts_seek_track
    PLP
    BNE     .begin_cmd
    LDY     MOTOR_TIME+1
    BPL     .begin_cmd
.on_time_delay:
    LDY     #$12
```

```
.on_time_delay_inner:
    DEY
    BNE      .on_time_delay_inner
    INC      MOTOR_TIME
    BNE      .on_time_delay
    INC      MOTOR_TIME+1
    BNE      .on_time_delay
.begin_cmd:
    LDY      #$0C
    LDA      (PTR2IOB),Y
    BEQ      .was_seek
    CMP      #$04
    BEQ      .was_format
    ROR
    PHP
    BCS      .reset_cnt
    JSR      PRENIBBLE
.reset_cnt:
    LDY      #$30
    STY      READ_CTR
.set_x_slot:
    LDX      SLOTPG5
    JSR      READ_ADDR
    BCC      .addr_read_good
.reduce_read_cnt:
    DEC      READ_CTR
    BPL      .set_x_slot
.do_recalibrate:
    LDA      CURR_TRACK
    PHA
    LDA      #$60
    JSR      rwts_set_track
    DEC      RECALIBCNT
    BEQ      .drive_err
    LDA      #$04
    STA      RESEEKCNT
    LDA      #$00
    JSR      rwts_seek_track
    PLA
.reseek:
    JSR      rwts_seek_track
    JMP      .reset_cnt
.addr_read_good:
    LDY      TRACK_ON_DISK
    CPY      CURR_TRACK
    BEQ      .found_track
    LDA      CURR_TRACK
    PHA
    TYA
    JSR      rwts_set_track
```

```
    PLA
    DEC     RESEEKCNT
    BNE     .reseek
    BEQ     .do_recalibrate
.drive_err:
    PLA
    LDA     #$40
.to_err_rwts:
    PLP
    JMP     .rwts_err
.was_seek:
    BEQ     .rwts_exit
.was_format:
    JMP     rwts_format
.found_track:
    LDY     #$03
    LDA     (PTR2IOB),Y
    PHA
    LDA     CHECKSUM_DISK
    LDY     #$0E
    STA     (PTR2IOB),Y
    PLA
    BEQ     .found_volume
    CMP     CHECKSUM_DISK
    BEQ     .found_volume
    LDA     #$20
    BNE     .to_err_rwts
.found_volume:
    LDY     #$05
    LDA     (PTR2IOB),Y
    TAY
    LDA     PHYSECTOR,Y
    CMP     SECTOR_DSK
    BNE     .reduce_read_cnt
    PLP
    BCC     .write
    JSR     READ
    PHP
    BCS     .reduce_read_cnt
    PLP
    LDX     #$00
    STX     RWTS_SCRATCH
    JSR     POSTNIBBLE
    LDX     SLOTPG5
.rwts_exit:
    CLC
    HEX     24      ; BIT instruction skips next SEC
.rwts_err:
    SEC
    LDY     #$0D
```

```
      STA     (PTR2IOB),Y
      LDA     MOTOROFF,X
      RTS
  .write:
      JSR     WRITE
      BCC     .rwts_exit
      LDA     #$10
      BCS     .rwts_err
```

Defines:
`RWTS_entry`, used in chunk 27.

Uses `ARM_MOVE_DELAY` 231, `PHYSECTOR` 247b, `POSTNIBBLE` 225b, `PRENIBBLE` 221, `READ` 226, `READ_ADDR` 228, `RWTS` 209, `WRITE` 223, `rwts_format` 243, `rwts_seek_track` 240, `rwts_set_track` 242, and `save_drive` 148b.

240 ⟨*RWTS seek track* 240⟩≡ (250)

```
  rwts_seek_track:
      ; Determines drive type and moves disk arm
      ; to desired track.
      SUBROUTINE

      PHA
      LDY     #$01
      LDA     (PTR2DCT),Y
      ROR
      PLA
      BCC     rwts_move_arm
      ASL
      JSR     rwts_move_arm
      LSR     CURR_TRACK
      RTS
```

Defines:
`rwts_seek_track`, used in chunks 236 and 243.

Uses `rwts_move_arm` 241a.

241a ⟨*RWTS move arm* 241a⟩≡ (250)

```
rwts_move_arm:
    ; Moves disk arm to desired track.
    SUBROUTINE

    STA      DEST_TRACK
    JSR      rwts_slot_x_to_y
    LDA      CURR_TRACK,Y
    BIT      ZPAGE_DRIVE
    BMI      .set_curr_track
    LDA      RESEEKCNT,Y
.set_curr_track:
    STA      CURR_TRACK
    LDA      DEST_TRACK
    BIT      ZPAGE_DRIVE
    BMI      .using_drive_1
    STA      RESEEKCNT,Y
    BPL      .using_drive_2
.using_drive_1:
    STA      CURR_TRACK,Y
.using_drive_2:
    JMP      SEEKABS
```

Defines:
  rwts_move_arm, used in chunk 240.
Uses SEEKABS 230 and rwts_slot_x_to_y 241b.

241b ⟨*RWTS Slot X to Y* 241b⟩≡ (250)

```
rwts_slot_x_to_y:
    ; Moves slot*16 in X to slot in Y.
    TXA
    LSR
    LSR
    LSR
    LSR
    TAY
    RTS
```

Defines:
  rwts_slot_x_to_y, used in chunks 241a and 242.

242 ⟨*RWTS set track* 242⟩≡ (250)

```
rwts_set_track:
    ; Sets track for RWTS.
    SUBROUTINE

    PHA
    LDY         #$02
    LDA         (PTR2IOB),Y
    ROR
    ROR         ZPAGE_DRIVE
    JSR         rwts_slot_x_to_y
    PLA
    ASL
    BIT         ZPAGE_DRIVE
    BMI         .store_drive_1
    STA         TRACK_FOR_DRIVE_2,Y
    BPL         .end
.store_drive_1:
    STA         TRACK_FOR_DRIVE_1,Y
.end:
    RTS
```

Defines:
`rwts_set_track`, used in chunks 236 and 243.

Uses `RWTS` 209 and `rwts_slot_x_to_y` 241b.

243 ⟨*RWTS Format disk* 243⟩≡ (250)

```
rwts_format:
    ; Formats a disk.
    SUBROUTINE

    LDY     #$03
    LDA     (PTR2IOB),Y
    STA     FORMAT_VOLUME
    LDA     #$AA
    STA     PTR2BUF
    LDY     #$56
    LDA     #$00
    STA     FORMAT_TRACK
.zbuf2:
    STA     PRIMARY_BUFF+255,Y
    DEY
    BNE     .zbuf2
.zbuf1:
    STA     PRIMARY_BUFF,Y
    DEY
    BNE     .zbuf1
    LDA     #$50
    JSR     rwts_set_track
    LDA     #$28
    STA     SYNC_CTR
.format_next_track:
    LDA     FORMAT_TRACK
    JSR     rwts_seek_track
    JSR     rwts_format_track
    LDA     #$08
    BCS     .format_err
    LDA     #$30
    STA     READ_CTR
.read_again:
    SEC
    DEC     READ_CTR
    BEQ     .format_err
    JSR     READ_ADDR
    BCS     .read_again
    LDA     SECTOR_DSK
    BNE     .read_again
    JSR     READ
    BCS     .read_again
    INC     FORMAT_TRACK
    LDA     FORMAT_TRACK
    CMP     #$23
    BCC     .format_next_track
    CLC
    BCC     .format_done
.format_err:
```

```
    LDY      #$0D
    STA      (PTR2IOB),Y
    SEC
.format_done:
    LDA      MOTOROFF,X
    RTS
```

Defines:
rwts_format, used in chunk 236.
Uses PRIMARY_BUFF 233a, READ 226, READ_ADDR 228, rwts_format_track 245, rwts_seek_track 240, and rwts_set_track 242.

245 ⟨*RWTS Format track* 245⟩≡ (250)

```
rwts_format_track:
    ; Formats a track.
    SUBROUTINE

    LDA     #$00
    STA     FORMAT_SECTOR
    LDY     #$80
    BNE     .do_addr
.format_sector:
    LDY     SYNC_CTR
.do_addr:
    JSR     WRITE_ADDR_HDR
    BCS     .return
    JSR     WRITE
    BCS     .return
    INC     FORMAT_SECTOR
    LDA     FORMAT_SECTOR
    CMP     #$10
    BCC     .format_sector
    LDY     #$0F
    STY     FORMAT_SECTOR
    LDA     #$30
    STA     READ_CTR
.fill_sector_map:
    STA     SECTOR_FLAGS,Y
    DEY
    BPL     .fill_sector_map
    LDY     SYNC_CTR
.bypass_syncs:
    JSR     .return
    JSR     .return
    JSR     .return
    PHA
    PLA
    NOP
    DEY
    BNE     .bypass_syncs
    JSR     READ_ADDR
    BCS     .reread_addr
    LDA     SECTOR_DSK
    BEQ     .read_next_data_sector
    LDA     #$10
    CMP     SYNC_CTR
    LDA     SYNC_CTR
    SBC     #$01
    STA     SYNC_CTR
    CMP     #$05
    BCS     .reread_addr
    SEC
```

```
        RTS
    .read_next_addr:
        JSR     READ_ADDR
        BCS     .bad_read
    .read_next_data_sector:
        JSR     READ
        BCC     .check_sector_map
    .bad_read:
        DEC     READ_CTR
        BNE     .read_next_addr
    .reread_addr:
        JSR     READ_ADDR
        BCS     .not_last
        LDA     SECTOR_DSK
        CMP     #$0F
        BNE     .not_last
        JSR     READ
        BCC     rwts_format_track
    .not_last:
        DEC     READ_CTR
        BNE     .reread_addr
        SEC
    .return:
        RTS
    .check_sector_map:
        LDY     SECTOR_DSK
        LDA     SECTOR_FLAGS,Y
        BMI     .bad_read
        LDA     #$FF
        STA     SECTOR_FLAGS,Y
        DEC     FORMAT_SECTOR
        BPL     .read_next_addr
        LDA     FORMAT_TRACK
        BNE     .no_track_0
        LDA     SYNC_CTR
        CMP     #$10
        BCC     .return
        DEC     SYNC_CTR
        DEC     SYNC_CTR
    .no_track_0:
        CLC
        RTS
```

Defines:
`rwts_format_track`, used in chunk 243.

Uses `READ` 226, `READ_ADDR` 228, `SECTOR_FLAGS` 247a, `WRITE` 223, and `WRITE_ADDR_HDR` 234.

247a ⟨*RWTS Sector flags* 247a⟩≡ (250)

```
SECTOR_FLAGS:
    HEX      FF FF FF FF FF FF FF FF
    HEX      FF FF FF FF FF FF FF FF
```

Defines:
SECTOR_FLAGS, used in chunk 245.

247b ⟨*RWTS Physical sector numbers* 247b⟩≡ (250)

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

Defines:
PHYSECTOR, used in chunk 236.

247c ⟨*RWTS Clobber language card* 247c⟩≡ (250)

```
RWTS_CLOBBER_LANG_CARD:
    SUBROUTINE

    JSR      SETVID
    LDA      PHASEON
    LDA      PHASEON
    LDA      #$00
    STA      $E000
    JMP      BACK_TO_BOOT2
    HEX      00 00 00
```

Defines:
RWTS_CLOBBER_LANG_CARD, never used.
Uses SETVID 208.

247d ⟨*RWTS Zero patch* 247d⟩≡ (250)

```
RWTS_ZERO_PATCH:
    SUBROUTINE

    STA      $1663
    STA      $1670
    STA      $1671
    RTS
```

Defines:
RWTS_ZERO_PATCH, never used.

247e ⟨*RWTS Patch 2* 247e⟩≡ (250)

```
RWTS_PATCH_2:
    SUBROUTINE

    JSR      $135B
    STY      $16B7
    RTS
```

Defines:
RWTS_PATCH_2, never used.

248 ⟨*RWTS Disk full error patch* 248⟩≡ (250)

```
RWTS_DISK_FULL_PATCH:
    SUBROUTINE

    JSR     $1A7E
    LDX     $1F9B
    TXS
    JSR     $0F16
    TSX
    STX     $1F9B
    LDA     #$09
    JMP     $1F85
```

Defines:
RWTS_DISK_FULL_PATCH, never used.

249 ⟨*RWTS defines* 249⟩≡ (206b)

```
PHASEOFF        EQU     $C080
PHASEON         EQU     $C081
MOTOROFF        EQU     $C088
MOTORON         EQU     $C089
DRV0EN          EQU     $C08A
DRV1EN          EQU     $C08B
Q6L             EQU     $C08C
Q6H             EQU     $C08D
Q7L             EQU     $C08E
Q7H             EQU     $C08F

CURR_TRACK      EQU     $0478
TRACK_FOR_DRIVE_1 EQU   $0478   ; reused
RESEEKCNT       EQU     $04F8
TRACK_FOR_DRIVE_2 EQU   $04F8   ; reused
READ_CTR        EQU     $0578
SLOTPG5         EQU     $05F8
SLOTPG6         EQU     $0678
RECALIBCNT      EQU     $06F8

RWTS_SCRATCH    EQU     $26
RWTS_SCRATCH2   EQU     $27
DEST_TRACK      EQU     $2A
SLOT16          EQU     $2B
CKSUM_ON_DISK   EQU     $2C
SECTOR_DSK      EQU     $2D
TRACK_ON_DISK   EQU     $2E
VOLUME_ON_DISK  EQU     $2F
CHECKSUM_DISK   EQU     $2F     ; reused
ZPAGE_DRIVE     EQU     $35
PTR2DCT         EQU     $3C     ; 2 bytes
PTR2BUF         EQU     $3E     ; 2 bytes
FORMAT_SECTOR   EQU     $3F     ; reused
FORMAT_VOLUME   EQU     $41
FORMAT_TRACK    EQU     $44
SYNC_CTR        EQU     $45
MOTOR_TIME      EQU     $46     ; 2 bytes
PTR2IOB         EQU     $48     ; 2 bytes
DEBUG_JUMP      EQU     $7C
SECTORS_PER_TRACK EQU   $7F
```

Uses `DEBUG_JUMP` 209 and `SECTORS_PER_TRACK` 209.

250 ⟨*RWTS routines* 250⟩≡ (30b)
⟨*RWTS Prenibble routine* 221⟩
⟨*RWTS Write routine* 223⟩
⟨*RWTS Write bytes* 225a⟩
⟨*RWTS Postnibble routine* 225b⟩
⟨*RWTS Read routine* 226⟩
⟨*RWTS Read address* 228⟩
⟨*RWTS Seek absolute* 230⟩
⟨*RWTS Arm move delay* 231⟩
⟨*RWTS Arm move delay tables* 232a⟩
⟨*RWTS Write translate table* 232b⟩
⟨*RWTS Unused area* 232c⟩
⟨*RWTS Read translate table* 232d⟩
⟨*RWTS Primary buffer* 233a⟩
⟨*RWTS Secondary buffer* 233b⟩
⟨*RWTS Write address header* 234⟩
⟨*RWTS Write address header bytes* 235a⟩
⟨*RWTS Unused area 2* 235b⟩
⟨*RWTS Entry point* 236⟩
⟨*RWTS seek track* 240⟩
⟨*RWTS move arm* 241a⟩
⟨*RWTS Slot X to Y* 241b⟩
⟨*RWTS set track* 242⟩
⟨*RWTS Format disk* 243⟩
⟨*RWTS Format track* 245⟩
⟨*RWTS Sector flags* 247a⟩
⟨*RWTS Physical sector numbers* 247b⟩
⟨*RWTS Clobber language card* 247c⟩
⟨*RWTS Zero patch* 247d⟩
⟨*RWTS Patch 2* 247e⟩
⟨*RWTS Disk full error patch* 248⟩

Chapter 19

Index