Ali Baba and the Forty Thieves

Reverse engineered from the Apple II version

Robert Baruch

Contents

Chapter 1

Ali Baba and the Forty Thieves

1.1 Introduction

Ali Baba and the Forty Thieves (Wikipedia link) is a graphical computer role-playing game (`CRPG`) written by Stuart Smith in 1981 and released in 1982 for the Apple II.

It was published by Quality Software. The game is a "hot-seat" multiplayer game, where players may take turns at the keyboard to control their character. The game can also be played single-player, with the player controlling all characters.

The goal is to rescue Princess Buddir and bring her back to Ali Baba's home. Forty thieves wander the dungeon, and there are other fixed enemies and wandering friendly `NPC`s who can join your party.

There is a good overview of the gameplay at CRPG Addict.

The purpose of this document is to reverse engineer the game. The disk image used is from the Internet Archive:

- Ali Baba and the Forty Thieves (4am and san inc crack)

Although originally published by Quality Software, the disk image shows that it was published by Electronic Arts. In 1986, Electronic Arts published Age of Adventure, a compilation of Ali Baba and Stuart Smith's The Return of Heracles (1983). This image is almost certainly from this compilation: 4am also cracked The Return of Heracles on the same day.

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 images extracted from the disk.

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.

Useful resources:

- Beneath Apple DOS, by Don Worth and Pieter Lechner, 1982.
- Apple II Computer Graphics, by Ken Williams, Bob Kernaghan, and Lisa Kernaghan, 1983.
- 6502 Assembly Language Programming, by Lance A. Leventhal, 1979.
- Beagle Bros Apple Colors and ASCII Values, Beagle Bros Micro Software Inc, 1984.
- Hi-Res Graphics and Animation Using Assembly Language, The Guide for Apple II Programmers, by Leanard I. Malkin, 1985.
- Understanding the Apple II, by Jim Sather, 1983.
- Apple II Monitors Peeled, Apple, 1981

1.3 The booting of a disk

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

In general, booting the typical Apple disk until it runs the actual program looks like this:

```
DISKCARD -> BOOT1 -> BOOT2 -> PROGRAM
```

The disk card has a small ROM whose purpose is to load `BOOT1`, which starts from track 0 sector 0, where the first byte of sector 0 tells the card how many sectors are in `BOOT1`. Standard `DOS 3.3`'s `BOOT1` is only one sector long, but other more custom disk loaders have more sectors (up to 16).

The purpose of `BOOT1` is to load `BOOT2`, which contains a set of general disk service routines. The entry point into `BOOT2` loads the main program and then jumps to it. The main program can then use the disk routines from `BOOT2`.

Since all we're interested in is the main program, we need to find out what happens at the beginning of `BOOT2`, which will tell us where the program is on the disk, and what the program's entry point is. If the program itself uses the disk for any reason, then we have to look at `BOOT2` more closely.

1.3.1 Sector interleaving

Within a single track, sectors are not necessarily stored in consecutive order. The reason is that it takes time for the program to process the data for a sector it just read. By the time the program is ready to read the next sector, the disk has rotated some amount.

It would make loading multiple sectors faster if the program "speed" and the disk "speed" were coordinated. Thus, for example, sector 1 might be placed half the disk around from sector 0. In standard `DOS 3.3`, sector 1 is placed nearly all the way around the disk – `7/8` of the way around!

1.4 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 0-4: `BOOT1`: Target address `$0800`, entry point `$0801`.
- Sector 16-47: `BOOT2`: Target address `$A000`, entry point `$A806`.

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

```
python -m extract --first 0 -n 5 \
  -i "Ali Baba and the Forty Thieves (4am and san inc crack).dsk" -o boot1.bin \
  --table boot1_xlat.txt
python -m extract --first 16 -n 32 \
  -i "Ali Baba and the Forty Thieves (4am and san inc crack).dsk" -o boot2.bin \
  --table boot2_xlat.txt
python -m extract --first 16 -n 26 \
  -i "Ali Baba and the Forty Thieves (4am and san inc crack).dsk" -o main.bin --skew
```

Chapter 2

6502 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

We use these macros 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).

7 ⟨*Macros 7*⟩≡ (224 225a) 8a ▷

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

Defines:
  STOW, used in chunks 54–56, 79, 93, 121, 124, 127, 131, 132, 137b, 139b, 141c, 143, 149, 154b, 162a, 166, 168, 170–74, 176b, 177a, and 223.

STOW2 does the same, but in the opposite order, still retaining little-endianness.

8a ⟨*Macros* 7⟩+≡ (224 225a) ◁7 8b▷

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

Defines:
STOW2, used in chunk 132.

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.

8b ⟨*Macros* 7⟩+≡ (224 225a) ◁8a 8c▷

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

Defines:
MOVB, used in chunks 62a, 69, 72b, 75a, 79, 92b, 95, 107a, 136, 150a, 152–54, and 204a.
STOB, used in chunks 23a, 54c, 55b, 59c, 61, 66, 67, 70, 72b, 73, 75b, 77, 79, 85, 86, 89, 92b, 93, 95, 97, 101b, 104, 119b, 127, 130, 136, 138d, 141a, 144a, 150a, 152a, 155, 156, 171, and 182a.

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`.

8c ⟨*Macros* 7⟩+≡ (224 225a) ◁8b 9a▷

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

Defines:
MOVW, used in chunks 56c, 66, 67, 70, 77, 80b, 85, 92b, 121, 124, 130, 136, 138d, 140, 150a, 152–55, 160b, 174b, 177b, 189b, 192b, 194–97, 199b, 200a, 202a, 203a, 205a, 206a, 208, 209, and 215.

2.6.3 PSHW, PULB, PULW

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

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

9a ⟨*Macros* 7⟩+≡ (224 225a) ◁8c 9b▷

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

Defines:
PSHW, used in chunks 77, 80b, 121, 124, 132, 147, 148, 158–60, 183a, and 219.

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

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

9b ⟨*Macros* 7⟩+≡ (224 225a) ◁9a 9c▷

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

Defines:
PULB, used in chunk 152b.

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

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

9c ⟨*Macros* 7⟩+≡ (224 225a) ◁9b 10a▷

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

Defines:
PULW, used in chunks 77, 80b, 121, 124, 132, 147, 148, 159, 160a, 183a, and 219.

2.6.4 INCW

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

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

10a ⟨*Macros* 7⟩+≡ (224 225a) ◁9c 10b▷

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

Defines:
INCW, used in chunks 64, 65, 131, 132, 160b, 161, 183a, 197a, and 216.

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

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

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

10b ⟨*Macros 7*⟩+≡ (224 225a) ◁10a 11a▷

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

Defines:
ADDA, used in chunks 114 and 154b.

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

11a ⟨*Macros 7*⟩+≡ (224 225a) ◁10b 11b▷

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

Defines:
ADDAC, used in chunk 216.

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`.

11b ⟨*Macros* 7⟩+≡ (224 225a) ◁11a 11c▷

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

Defines:
ADDB, used in chunks 168 and 170b.

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

11c ⟨*Macros* 7⟩+≡ (224 225a) ◁11b 12a▷

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

Defines:
ADDB2, used in chunks 115 and 116.

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`.

12a ⟨*Macros* 7⟩+≡ (224 225a) ◁11c 12b▷

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

Defines:
ADDW, used in chunks 97, 113, 145, 147, 163, 188–91, and 193b.

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.

12b ⟨*Macros* 7⟩+≡ (224 225a) ◁12a 13a▷

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

Defines:
ADDWC, used in chunk 121.

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.

13a ⟨*Macros 7*⟩+≡ (224 225a) ◁12b 13b▷

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

Defines:
SUBB, used in chunks 63, 116, 154c, 183b, 186b, and 205a.

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

13b ⟨*Macros 7*⟩+≡ (224 225a) ◁13a 14a▷

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

Defines:
SUBB2, used in chunk 115b.

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.

14a ⟨*Macros* 7⟩+≡ (224 225a) ◁13b 14b▷

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

Defines:
SUBW, used in chunks 117b, 197c, and 216.

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.

14b ⟨*Macros* 7⟩+≡ (224 225a) ◁14a 15a▷

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

Defines:
SUBWL, used in chunk 118.

2.6.7 ROLW, RORW

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

15a ⟨*Macros* 7⟩+≡ (224 225a) ◁14b 15b▷

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

Defines:
`ROLW`, used in chunk 124.

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

15b ⟨*Macros* 7⟩+≡ (224 225a) ◁15a

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

Defines:
`RORW`, used in chunk 121.

Chapter 3

The boot process

Although the 4am crack of the disk is copyable and deployable as a standard `DOS 3.3` image, so there's no funny business about custom sector layouts and custom prologues and epilogues, nevertheless everything else about `BOOT1` and `BOOT2` has been left intact. This includes any decryption routines and other such protection schemes that do not involve the disk.

In this case, the game was distributed by Electronic Arts, Thus, it is important to understand what `BOOT`

3.1 BOOT1

After the disk card reads `BOOT1`, the zero-page location `IWMDATAPTR` is left as the pointer to memory just after `BOOT1`. The location `IWMSLTNDX` is the disk card's slot index (slot times 16).

16 ⟨*BOOT1* 16⟩≡ (224a)

Byte	Opcode	Argument	Operation
00	TJMP	addr	`IP <- addr`
01	CALL1	addr	`A <- addr(A)`
02	TBEQ	addr	If `A == 0`, `IP <- addr`
03	LDI	val	`A <- val`
04	LD	addr	`A <- (addr)`
05	TCALL	addr	Push IP onto stack, jump to addr
06	ST	addr	`(addr) <- A`
07	SUBI	val	`A -= val`
08	CALL0	addr	`addr()`
09	TRET	none	Pop IP from stack
0A	LDX	addr	`A <- (addr + A)`
0B	ASL	none	`A *= 2`
0C	INC	addr	`A <- ++(addr)`
0D	ADD	val	`A += val`
0E	DXR	none	decrypts (src)
0F	TBNE	addr	If `A != 0`, `IP <- addr`
10	SUB	addr	`A -= (addr)`
11	COPY	none	`(dest++) <- (src++)`

3.2 BOOT2

`BOOT2` contains a kind of mini virtual machine. The machine has a one-byte "psuedo-accumulator" (stored at `$48`), a two-byte pointer (`boot2_ptr`), and an instruction pointer. The machine has 18 opcodes (see table).

The addresses and values in the program are "encrypted". Addresses need to be exclusive-ored with `#$D903` and values need to be exclusive-ored with `#$4C`.

Here is the program:

17 ⟨*BOOT2 Program* 17⟩≡

```
A851:   LD      C050    ; TXTCLR soft switch (display graphics)
        LD      C052    ; MIXCLR (display full screen)
        LD      C057    ; HIRES (display hi-res graphics)
        LDI     F8
        ST      004C    ; boot2_src
        LDI     48
        ST      03F2
        LDI     A9
        ST      03F3    ; RESET handler = A948
        LDI     0C
        ST      03F4    ; Power-up byte
A86E:   LDI     FF
        CALL    FCA8    ; WAIT
        INC     004C    ; boot2_src
        TBEQ    A87C v
```

```
        JMP     A86E ^

 A87C:  LD      A805    ; it's 00.
        TBEQ    A88D v
        LDI     00
        TCALL   A967 v  ; Read track by index
        LD      C0E8    ; IWM: Motor off
        CALL1   0000

        ; Read tracks 7-C to 0x4000-0x95FF.
        ; Track B gets written to 0x8000-0x8FFF, but then
        ; track C gets written to 0x8600-0x95FF.
 A88D:  LDI     05
        TCALL   A967 v  ; Read track by index (Tracks 7-C)
        CALL1   A898    ; Zero out $400-$7FF
        JMP     A8B2 v

 A898:  ; machine language subroutine to zero out $400-$7FF (text page 0)

 A8B2:  LD      A800    ; it's 01
        TBEQ    A8BB v

 A8B8:  LD      C056    ; LORES (display lo-res)

        ; Read tracks:
        ;   0D -> 0x0800-0x17FF
        ;   0E -> 0x1000-0x1FFF (overwrites half of track 0D)
        ;   0F -> 0x2000-0x2FFF
        ;   10 -> 0x3000-0x3FFF
 A8BB:  LDI     0C
        TCALL   A967 v  ; Read track by index (starts with track 0D)

        ; Silly checksum routine on A000-A2??
        LDI     00
        ST      004C    ; boot2_src
        ST      A9F2    ; boot2_tmp1

 A8C8:  LD      004C    ; boot2_src
        LDX     A000
        SUB     A9F2
        ST      A9F2    ; boot2_tmp1 -= A000[boot2_src]
        LD      004C
        LDX     A100
        SUB     A9F2
        ST      A9F2    ; boot2_tmp1 -= A100[boot2_src]
        INC     004C
        TBNE    A8C8 ^

 A8E6:  LD      004C
        LDX     A200
```

```
        SUB     A9F2
        ST      A9F2    ; boot2_tmp1 -= A200[boot2_src]
        INC     004C
        SUBI    E0      ; A -= 0xE0
        TBNE    A8E6 ^

        LD      A9F2
        SUBI    60      ; A = boot2_tmp1 - 0x60
        TBNE    A9AB v  ; error (noreturn)

        LDI     05
        CALL1   BC00    ; jumps to BCD4 (move_arm_to_track)
        LD      A000
        LDI     00
        LD      C0E8    ; IWM: Motor off

        LDI     00
        ST      004C
        LDI     40
        ST      004D    ; boot2_src <- 4000

        LDI     02
        ST      AC60    ; boot2_bot <- 02
        LDI     03
        ST      AC61    ; boot2_top <- 03

A923:   CALL1   A970    ; funny-increment AC60
        DXR             ; decrypt byte at boot2_src
        TBNE    A923 ^

        ; Copy A300-A5FF to 0500-07FF
        LDI     00      ; boot2_src <- A300, boot2_dest <- 0500
        ST      004C
        ST      004E
        LDI     A3
        ST      004D
        LDI     05
        ST      004F

A93C:   COPY
        LD      004F
        SUBI    08
        TBNE    A93C ^

        CALL0   0800    ; call main!

A948:   ; machine language subroutine

        ; Read tracks by index, until track page is 0.
A967:   ST      A9F1    ; boot2_tmp0
```

```
        LD      C0E9    ; IWM: Motor on
        TJMP    A985 v

A970:   ; machine language subroutine

A985:   LD      A9F1    ; boot2_tmp0
        LDX     A9F3    ; boot2_table1 (track page)
        TBEQ    A9F0 v  ; Return
        ST      003E    ; boot2_track_page
        LD      A9F1    ; boot2_tmp0
        LDX     AA04    ; boot2_table2 (track)
        CALL1   BC03    ; jumps to BF00 (read track)
        TBEQ    A9A0 v  ; read track ok?
        TJMP    A9D4 v  ; Error routine (noreturn)

A9A0:   LD      A9F1    ; boot2_tmp0++
        SUBI    FF
        ST      A9F1
        TJMP    A985 ^

; Error routine: print '?' and turn motor on and off in a loop.
A9AB:   LD      C0E9    ; IWM: motor on
        CALL1   FC58    ; HOME (clear screen)
        LD      C051    ; TXTSET (display text screen)
        LDI     BF      ; '?'
        ST      0400    ; Display '?'
        LDI     40
        CALL1   FCA8    ; WAIT
        LD      C0E8    ; IWM: Motor off
A9C1:   LDI     60
        CALL1   FCA8    ; WAIT
        LDI     08
        CALL1   FCA8    ; WAIT
        INC     A9F2
        TBNE    A9C1 ^
        TJMP    A9AB ^

; Error routine: beep, print "ERR", turn off disk motor, and spinloop.
A9D4:   CALL1   FC58    ; HOME
        CALL1   FBDD    ; BELL
        LD      C051    ; TXTSET (display text screen)
        LDI     C5      ; 'E'
        ST      0400    ; text page 0 char 0
        LDI     D2      ; 'R'
        ST      0401    ; text page 0 char 1
        ST      0402    ; text page 0 char 2
        LD      C0E8    ; IWM: Motor off
A9ED:   TJMP    A9ED -  ; Spinloop

A9F0:   TRET
```

Uses HOME 226 and main 53a.

21a ⟨*BOOT2 track pages* 21a⟩≡

```
BOOT2_track_pages:
    HEX     08 10 20 30 00
    HEX     40 50 60 70 80 86 00
    HEX     08 10 20 30 00

BOOT2_tracks:
    HEX     03 04 21 22 00
    HEX     07 08 09 0A 0B 0C 00
    HEX     0D 0E 0F 10 00
```

Defines:
BOOT2_track_pages, never used.
BOOT2_tracks, never used.

21b ⟨*BOOT2 subroutine* 21b⟩≡

```
FUN_a970:
    INC      boot2_bot
    LDA      boot2_bot
    CMP      boot2_top
    BEQ      .inc
    RTS
.inc:
    LDA      #$01
    STA      boot2_bot
    INC      boot2_top
    RTS
```

21c ⟨*BOOT2 virtual DXR* 21c⟩≡

```
boot2_routine_0E_DXR_jmp:
    SUBROUTINE

    LDX      #$00           ; boot2_src ^= BOOT2_psuedoacc
    LDA      (boot2_src,X)
    EOR      BOOT2_psuedoacc
    STA      (boot2_src,X)
    INC      boot2_src      ; boot_src += 2
    INC      boot2_src
    BNE      .continue
    INC      boot2_src+1

.continue:
    LDA      boot2_src+1    ; BOOT2_psuedoacc = boot2_src_H ^ 0x68
    EOR      #$68
    STA      BOOT2_psuedoacc
    JMP      .loop
```

Defines:
boot2_routine_0E_DXR_jmp, never used.

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.

22 ⟨*BOOT2* 22⟩≡ (224b) 23a ▷

```
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 227, RWTS_entry 250, boot2_iob 24a, and sector_count 23a.

23a ⟨*BOOT2* 22⟩+≡ (224b) ◁22 23b▷

```
  .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 22.
Uses `DEBUG_JUMP` 227, `INIT` 226, `SECTORS_PER_TRACK` 227, `SETKBD` 226, `SETVID` 226, `STOB` 8b, and `main` 53a.

A zeroed out area:

23b ⟨*BOOT2* 22⟩+≡ (224b) ◁23a 24a▷

```
  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):

24a ⟨*BOOT2* 22⟩+≡ (224b) ◁23b 24b▷

```
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 22.
- 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 24b.

The Device Characteristics Table:

24b ⟨*BOOT2* 22⟩+≡ (224b) ◁24a 25a▷

```
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 24a.

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

25a ⟨*BOOT2* 22⟩+≡ (224b) ◁24b 25b▷

```
        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
```

25b ⟨*BOOT2* 22⟩+≡ (224b) ◁25a

⟨*RWTS routines* 264⟩

Chapter 4

Startup

As part of the startup routine, there are two blocks which are copied to higher areas of memory that had been occupied by `BOOT2`. These copies are from `2000-32FF` to `9600-A8FF`, and `3300-3FFF` to `B300-BFFF`.

Chapter 5

Apple II Graphics

Hi-res graphics on the Apple II is odd. Graphics are memory-mapped, not exactly consecutively, and bits don't always correspond to pixels. Color especially is odd, compared to today's luxurious 32-bit per pixel RGBA.

The Apple II has two hi-res graphics pages, and maps the area from `$2000-$3FFF` to high-res graphics page 1 (HGR1), and `$4000-$5FFF` to page 2 (HGR2).

5.1 Pixels and their color

First we'll talk about pixels. Nominally, the resolution of the hi-res graphics screen is 280 pixels wide by 192 pixels tall. In the memory map, each row is represented by 40 bytes. The high bit of each byte is not used for pixel data, but is used to control color.

Here are some rules for how these bytes are turned into pixels:

- Pixels are drawn to the screen from byte data least significant bit first. This means that for the first byte bit 0 is column 0, bit 1 is column 1, and so on.
- A pattern of `11` results in two white pixels at the `1` positions.
- A pattern of `010` results at least in a colored pixel at the `1` position.
- A pattern of `101` results at least in a colored pixel at the `0` position.
- So, a pattern of `01010` results in at least three consecutive colored pixels starting from the first `1` to the last `1`. The last `0` bit would also be colored if followed by a `1`.

- Likewise, a pattern of 11011 results in two white pixels, a colored pixel, and then two more white pixels.
- The color of a 010 pixel depends on the column that the 1 falls on, and also whether the high bit of its byte was set or not.
- The color of a 11011 pixel depends on the column that the 0 falls on, and also whether the high bit of its byte was set or not.

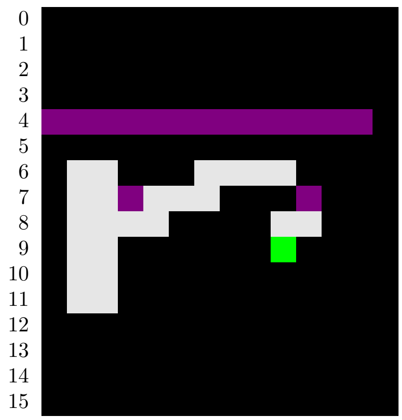
	Odd	Even
High bit clear	Green	Violet
High bit set	Orange	Blue

The implication is that you can only select one pair of colors per byte.

An example would probably be good here. We will take one of the font characters from the game, the lower-case `r`:

Bytes	Bits	Colorset	Pixel Data
00 00	00000000 00000000	0	00000000000000
00 00	00000000 00000000	0	00000000000000
00 00	00000000 00000000	0	00000000000000
00 00	00000000 00000000	0	00000000000000
55 2A	01010101 00101010	0	10101010101010
00 00	00000000 00000000	0	00000000000000
46 07	01000110 00000111	0	01100011110000
76 08	01110110 00001000	0	01101110001000
00 1E	00000000 00011110	0	01111000011000
00 06	00000000 00000110	0	01100000010000
00 06	00000000 00000110	0	01100000000000
00 06	00000000 00000110	0	01100000000000
2A 00	00101010 00000000	0	00000000000000
00 00	00000000 00000000	0	00000000000000
07 00	00000111 00000000	0	00000000000000
08 00	00001000 00000000	0	00000000000000

Assuming that the following bits are all zero, and we place the character starting at column 0, we should see this:



Here is a screenshot of that character, cut from the splash screen:

The violet line goes all the way across the character in that image because it is followed by another font character with that line, so the lines connect according to the pixel color rules.

5.1.1 The Hi-Res Character Generator

The Hi-Res Character Generator (HRCG) is a library provided by the Applesoft Toolkit which allows a developer to display a custom font. Applesoft Toolkit also contains Animatrix, which is a utility used to create fonts for the HRCG.

The manual for the Applesoft Toolkit can be found on the Internet Archive.

29 ⟨*init hcrg* 29⟩≡

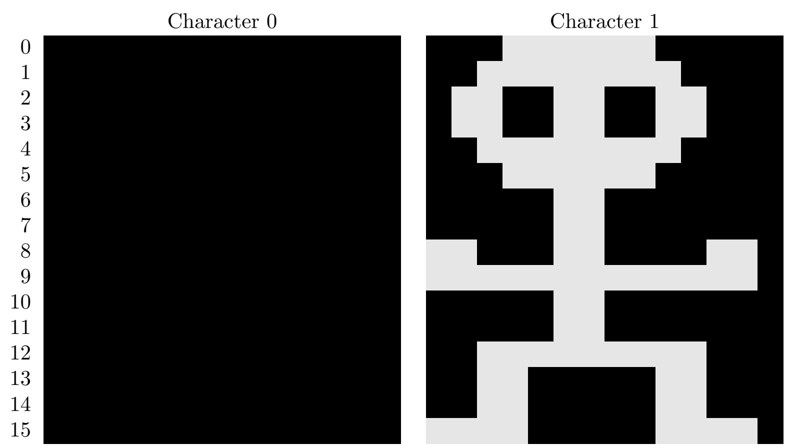
```
init_HRCG:
    SUBROUTINE

    ; Overwrites part of HRCG so it returns immediately after setting TXTPAGE1
    ; instead of continuing on to set TXTPAGE2.
    LDA     #$60                        ; RTS
    STA     $94B6

    ; Overwrites part of HRCG so that it does not call DOS_VEC_RECONNECT upon
    ; initialization.
    LDA     #$EA                        ; NOP
    STA     HRCG_overwritten_with_NOP
    STA     HRCG_overwritten_with_NOP+1
    STA     HRCG_overwritten_with_NOP+2
    JMP     HRCG_link_and_set_mode_jmp
```

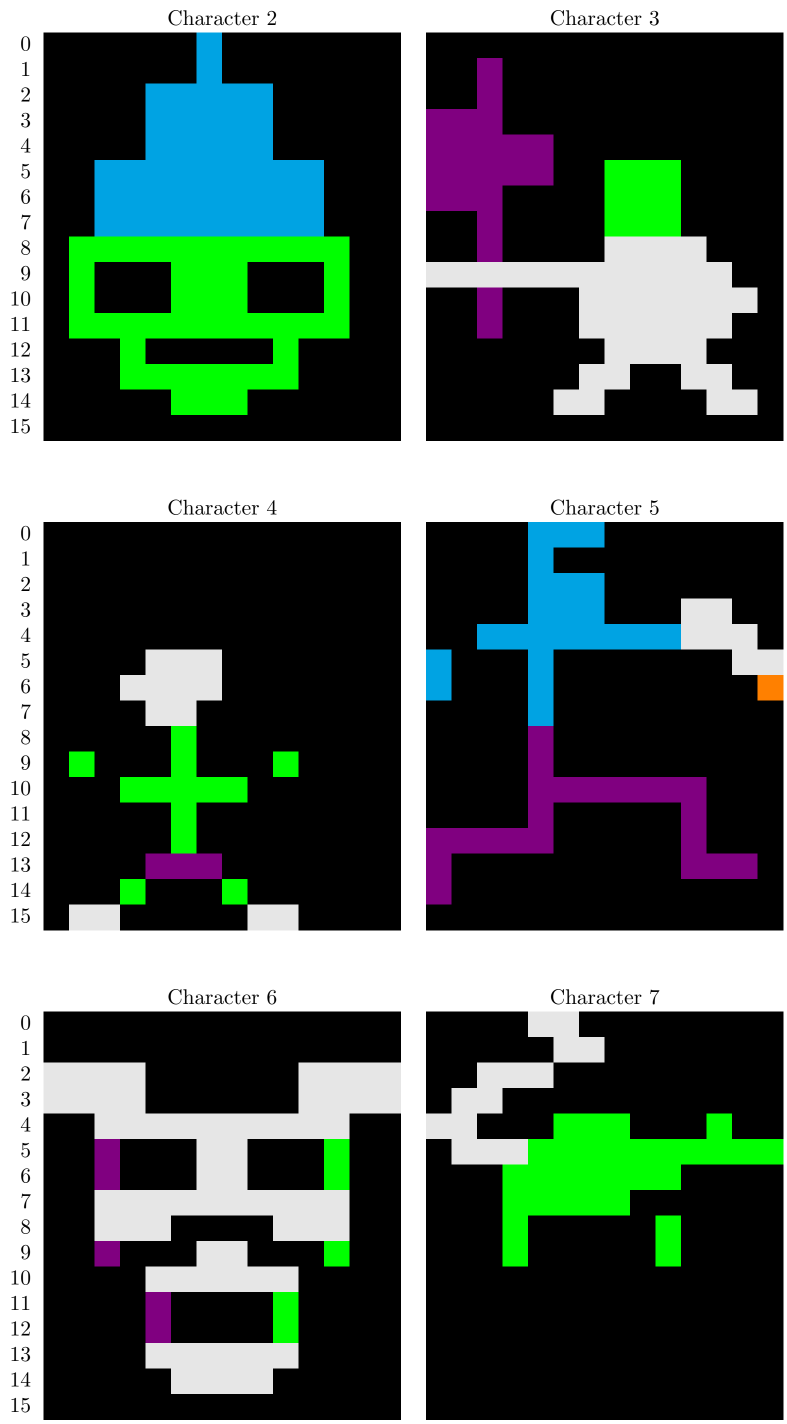
The new output handler installed by the HRCG is called whenever `COUT` is called. `A` contains the character to output.

The HRCG defines 96 characters as a character set. These characters are the "printing" characters, which are ASCII 32 (space) through 127 (delete). Character set 0 is always the standard set of Apple text characters. The font data for this standard set is stored at `$97A5-$9AA4`, consisting of 8 bytes per character. Each byte represents 7 consecutive pixels since the high bit is used for color set selection.

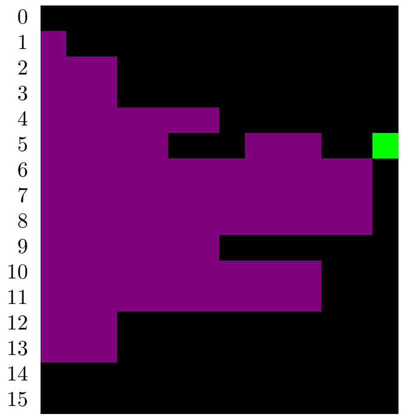
The Ali Baba font data for HRCG is stored at `$83A5-$92A4`, starting with character set 1. Each character is 32 bytes, broken up into four blocks of eight bytes (7x8 blocks) each. The first block is the upper left of the character, the second block is the upper right, the third is the lower left, and the fourth is the lower right. Thus, each character is 14x16.

Here are the font characters. There are 120 defined (which is more than the usual 96).

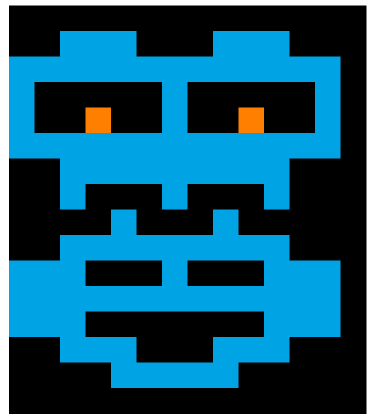



Character 2
Character 3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Character 4
Character 5
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Character 6
Character 7
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

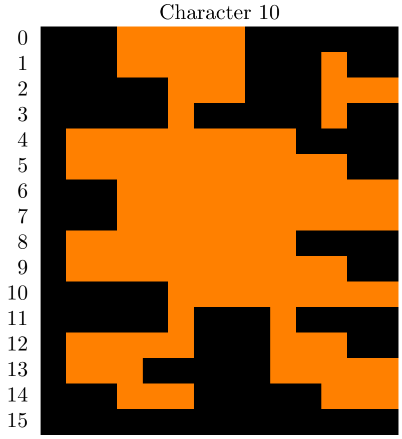
Character 8



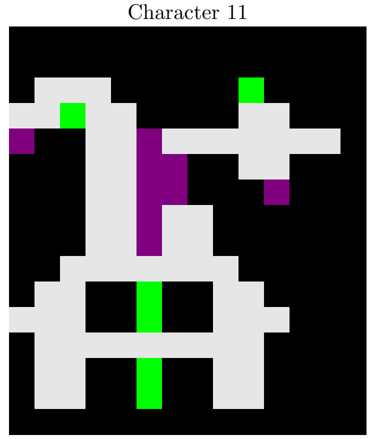
Character 9



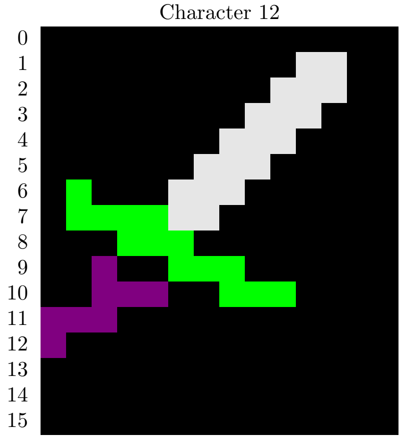
Character 10



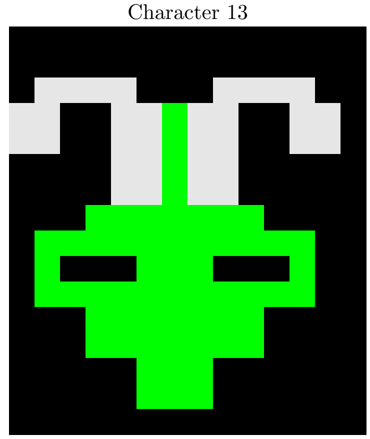
Character 11



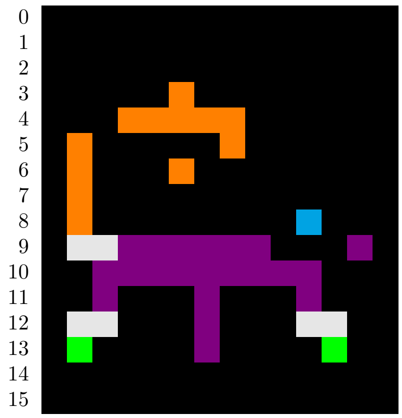
Character 12



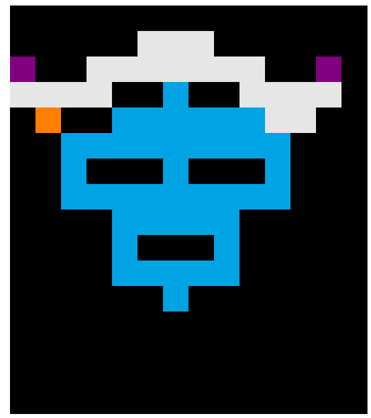
Character 13



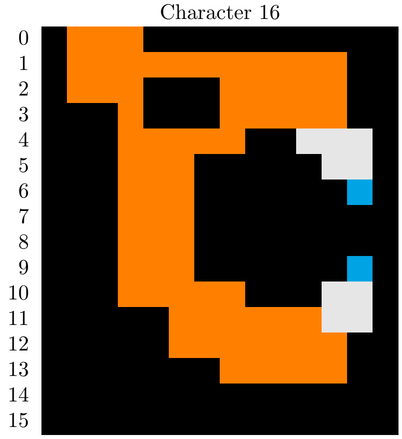
Character 14



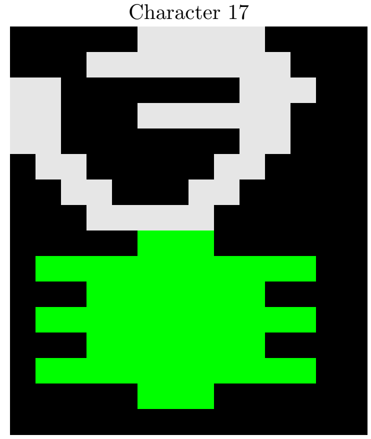
Character 15



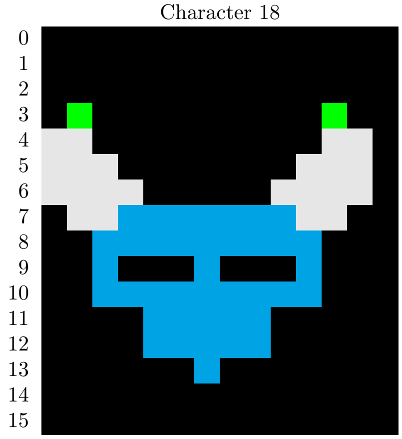
Character 16



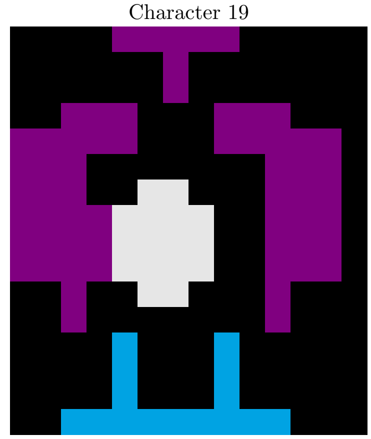
Character 17



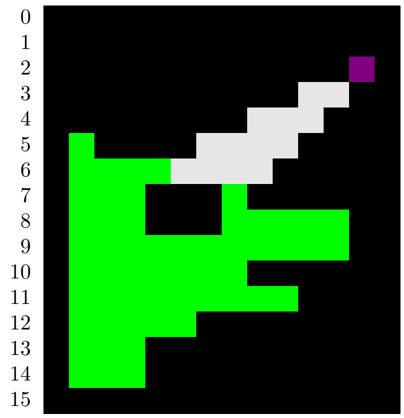
Character 18



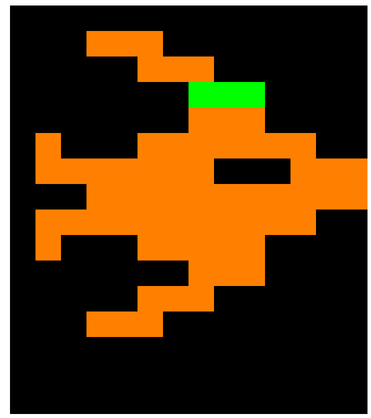
Character 19



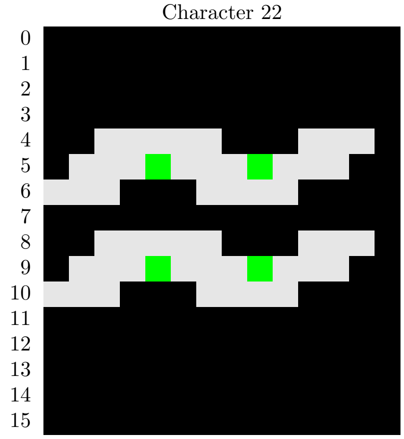
Character 20



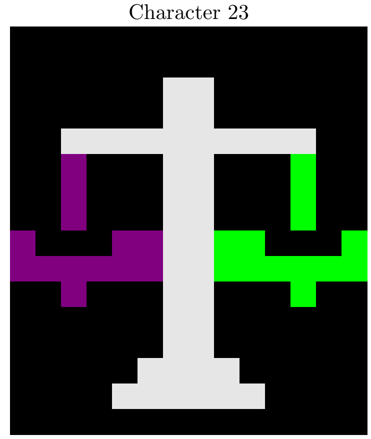
Character 21



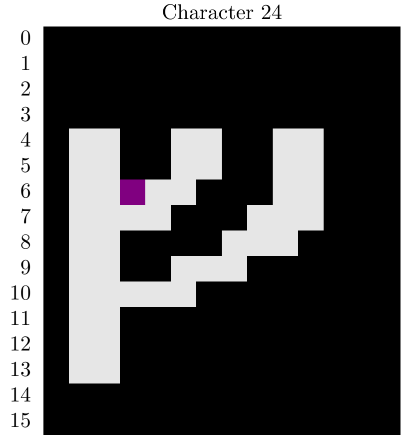
Character 22



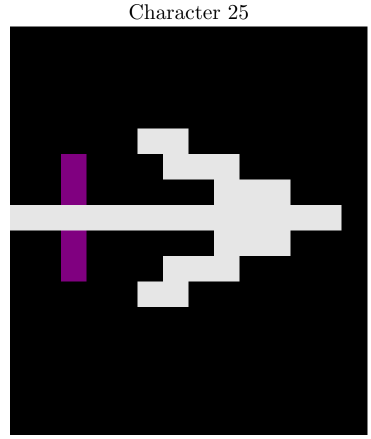
Character 23



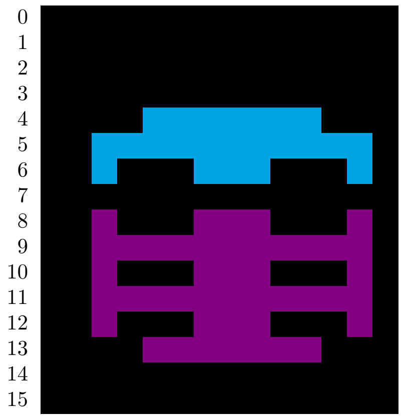
Character 24



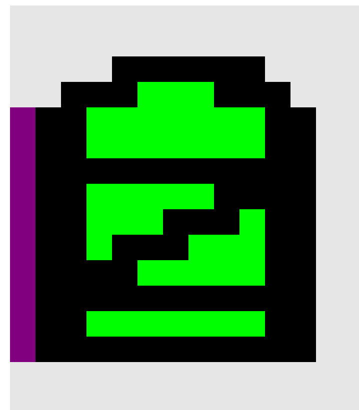
Character 25



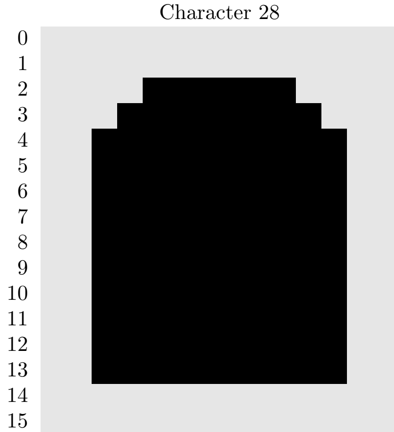
Character 26



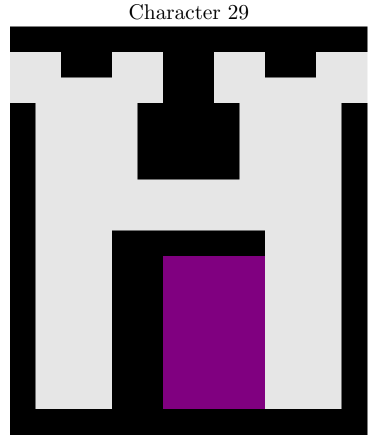
Character 27



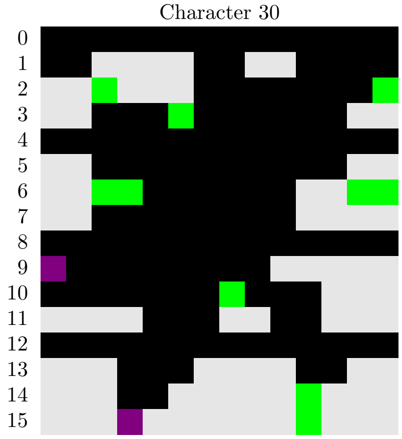
Character 28



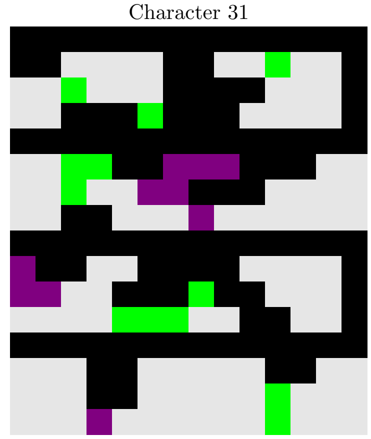
Character 29



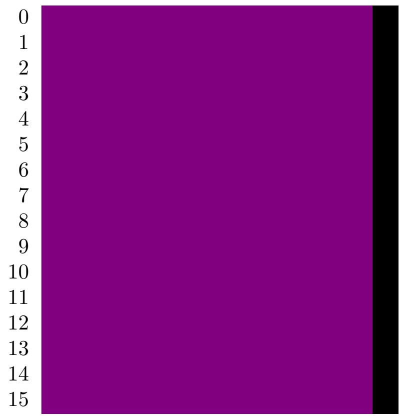
Character 30



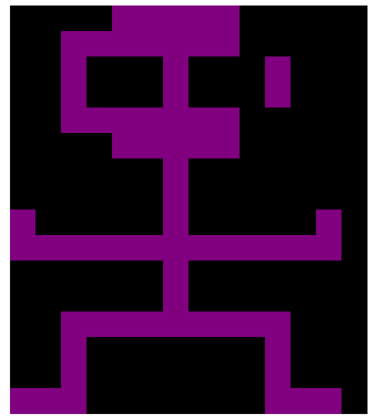
Character 31



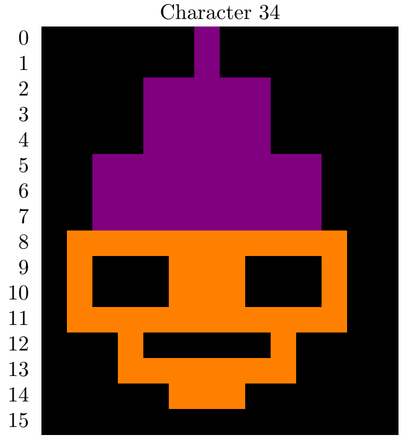
Character 32



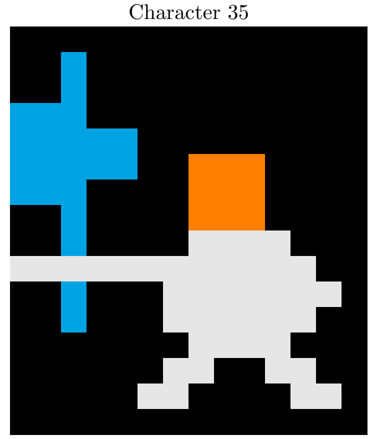
Character 33



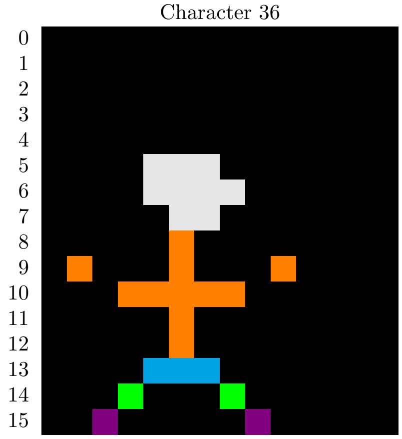
Character 34



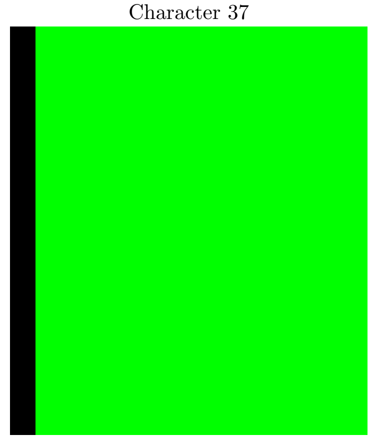
Character 35



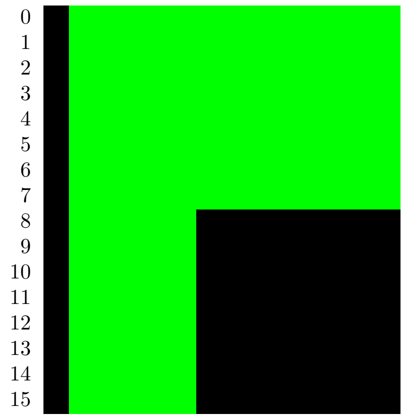
Character 36



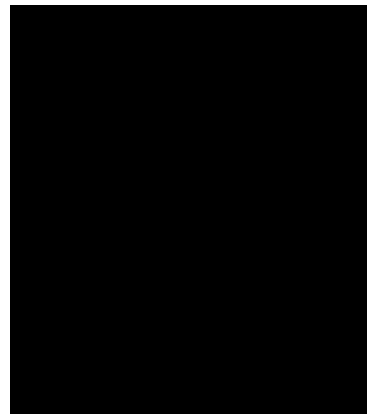
Character 37



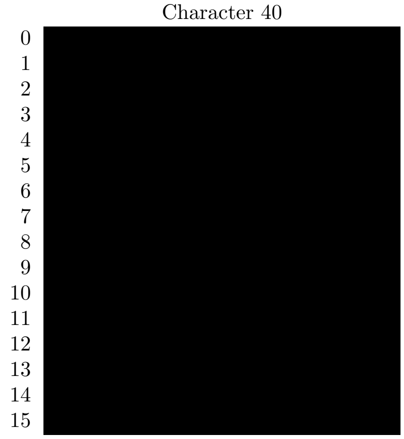
Character 38



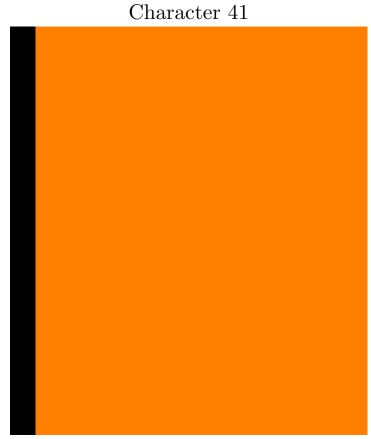
Character 39



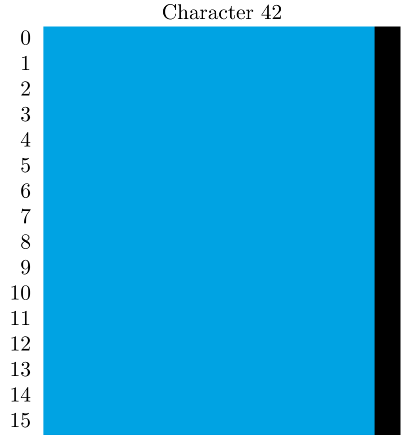
Character 40



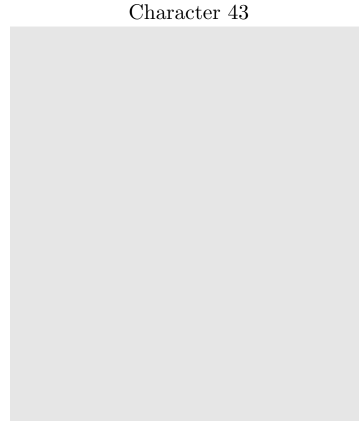
Character 41



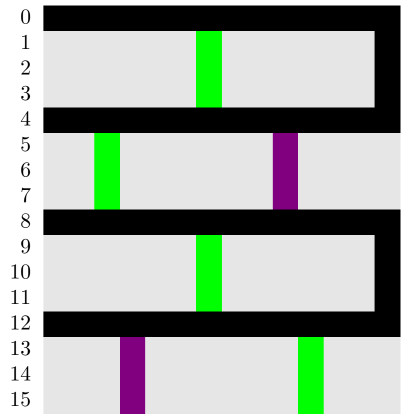
Character 42



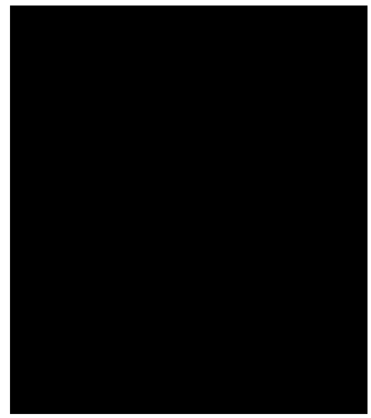
Character 43



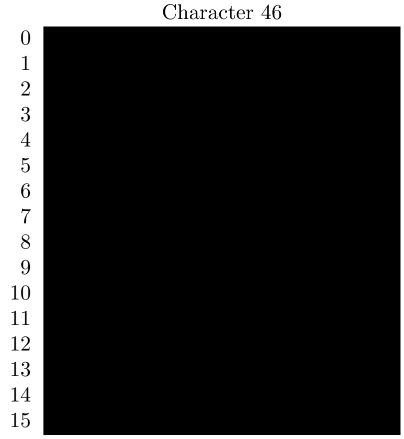
Character 44



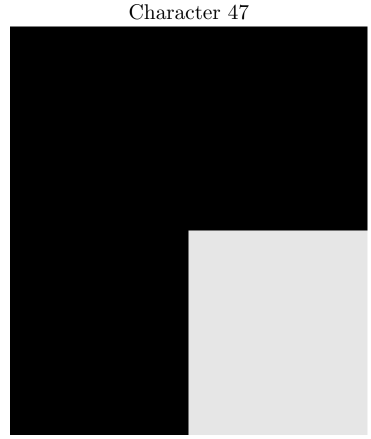
Character 45



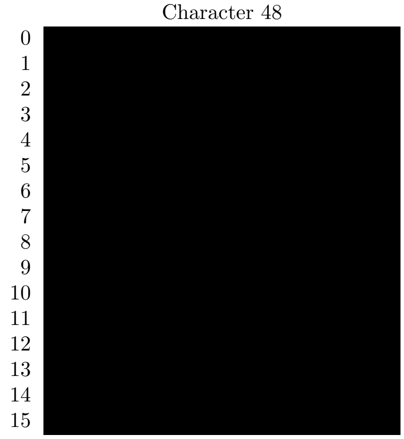
Character 46



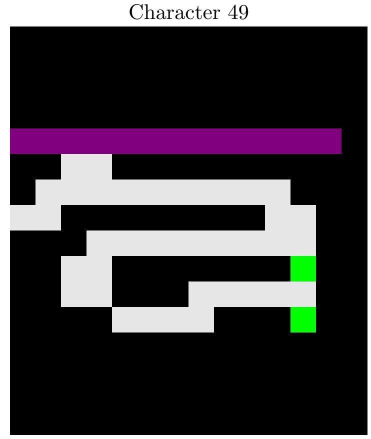
Character 47



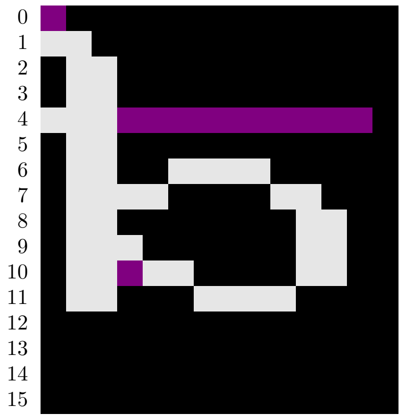
Character 48



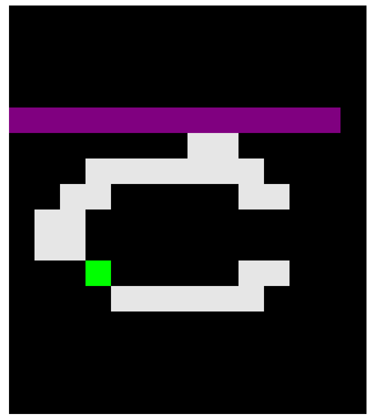
Character 49



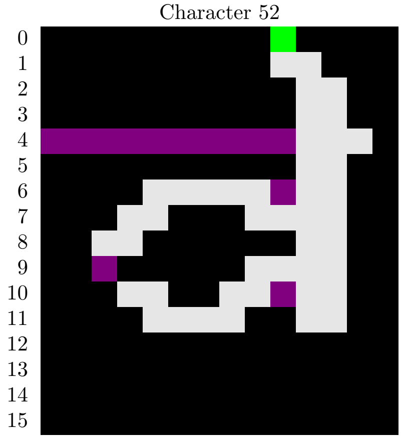
Character 50



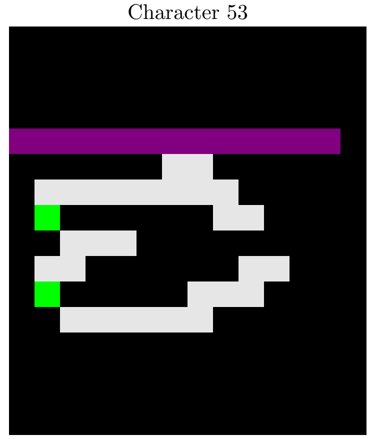
Character 51



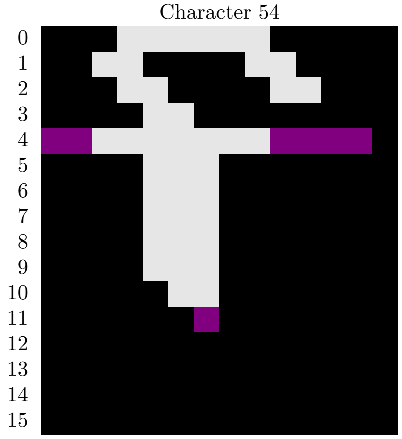
Character 52



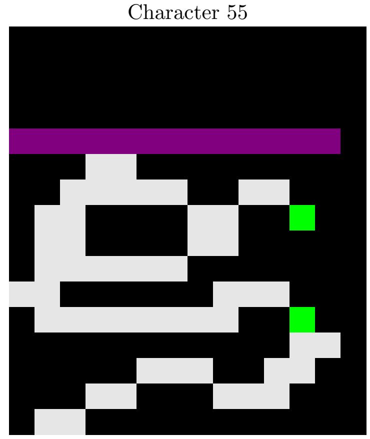
Character 53



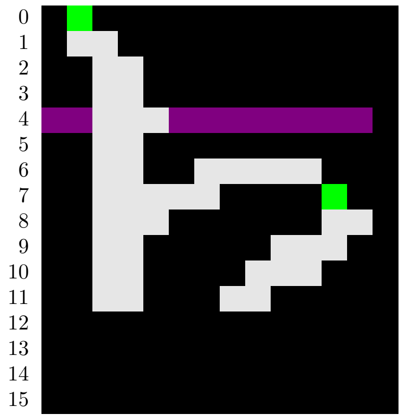
Character 54



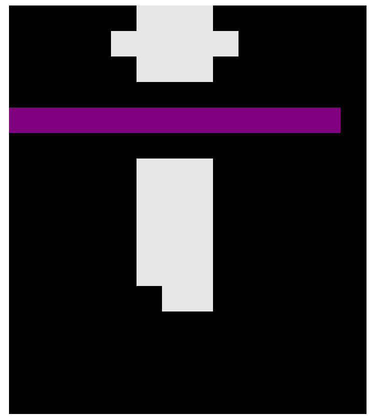
Character 55



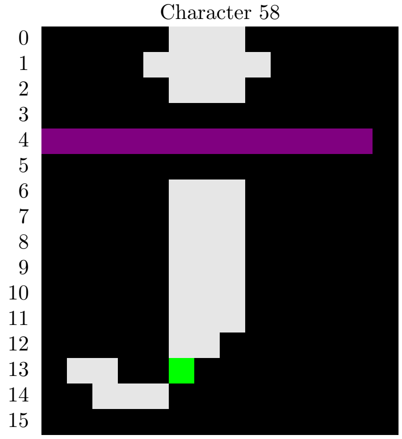
Character 56



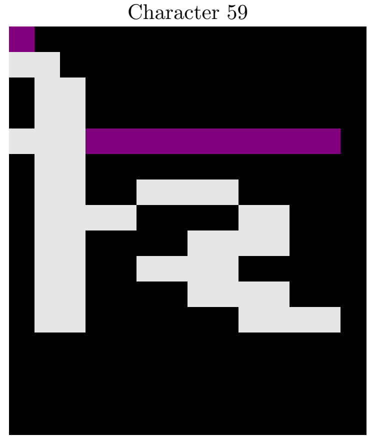
Character 57



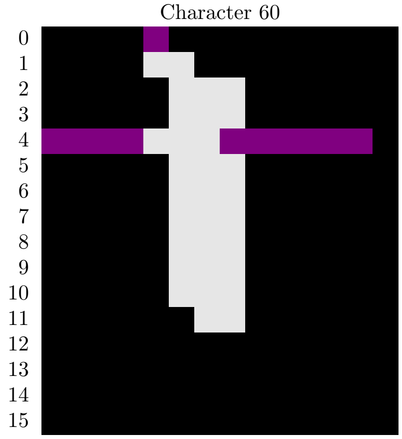
Character 58



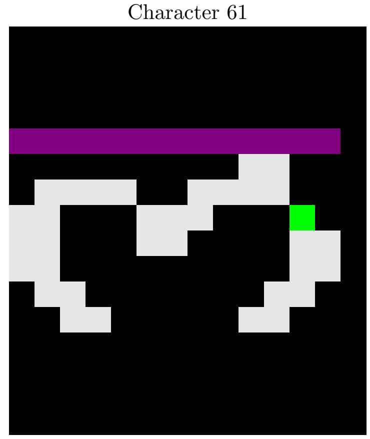
Character 59



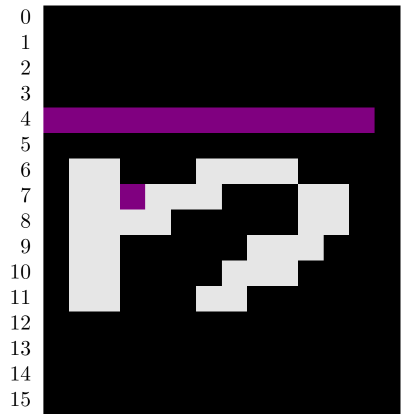
Character 60



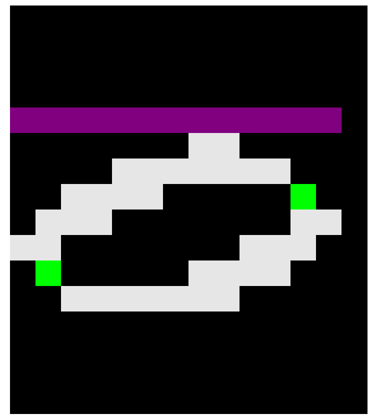
Character 61



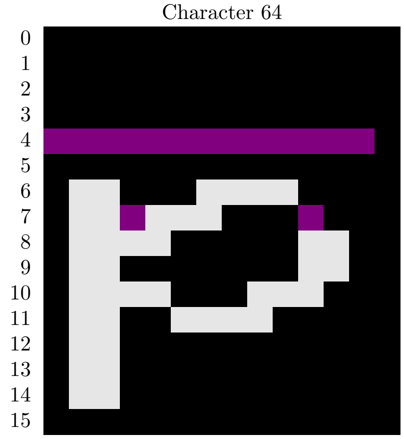
Character 62



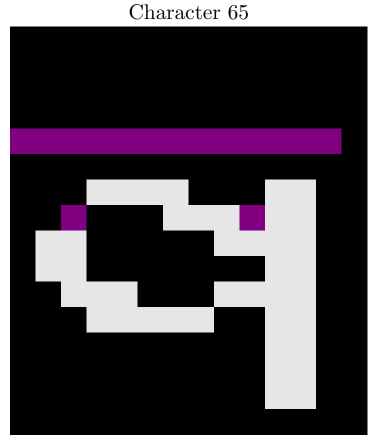
Character 63



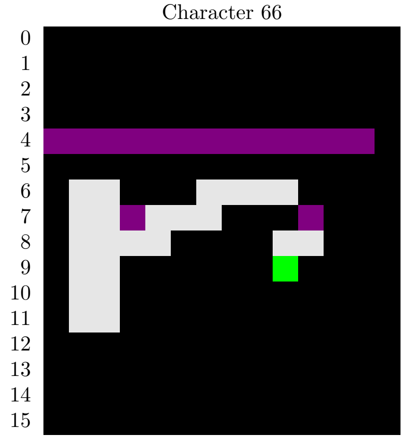
Character 64



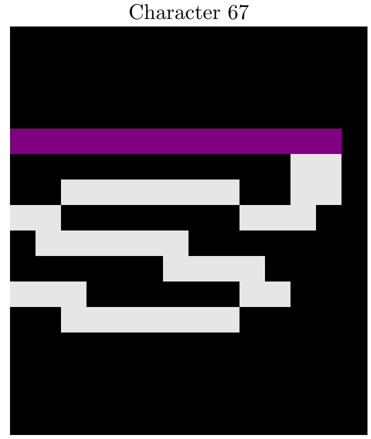
Character 65



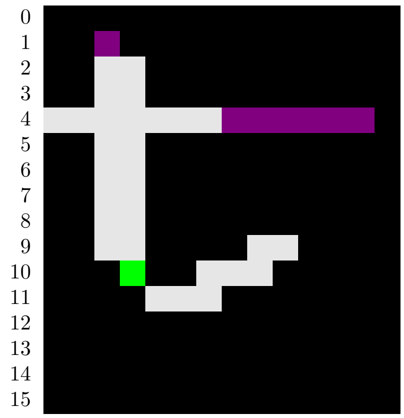
Character 66



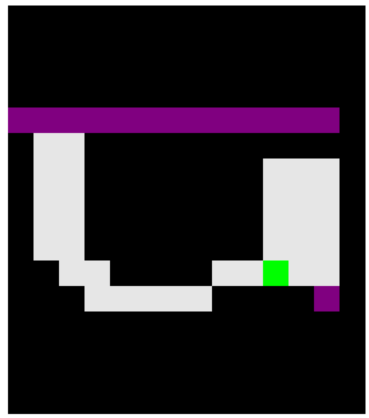
Character 67



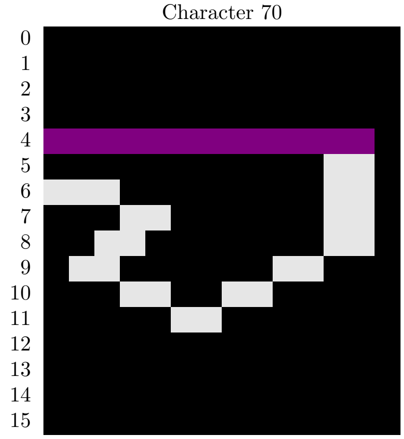
Character 68



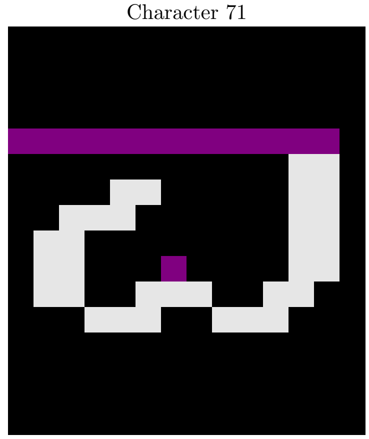
Character 69



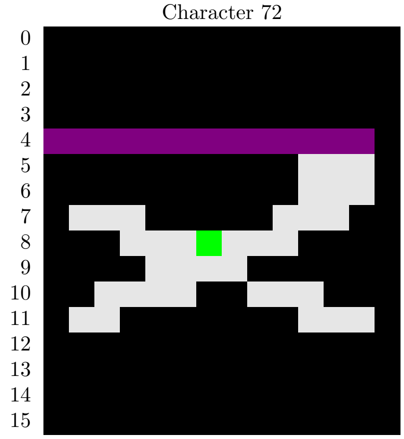
Character 70



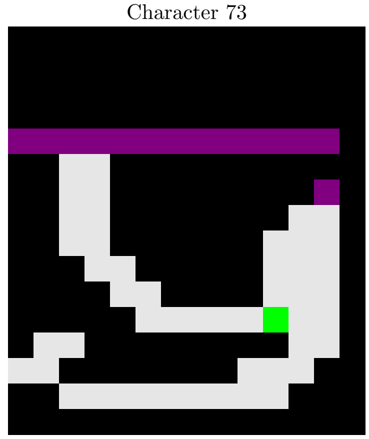
Character 71



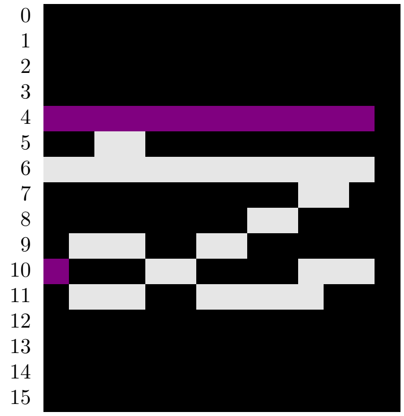
Character 72



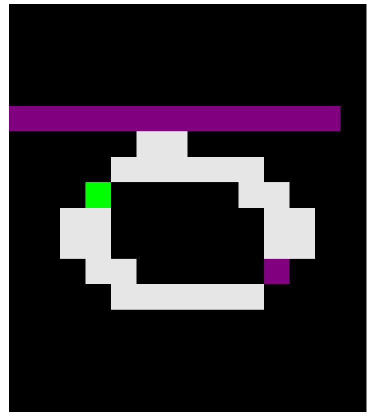
Character 73



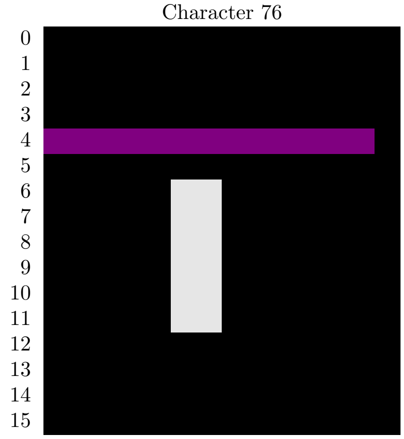
Character 74



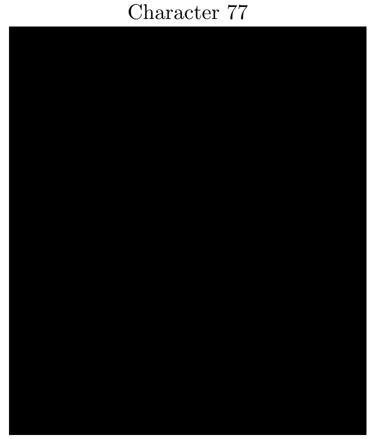
Character 75



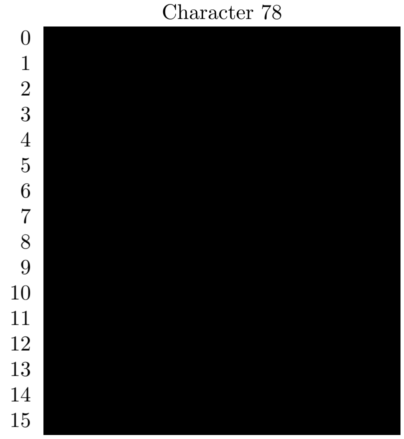
Character 76



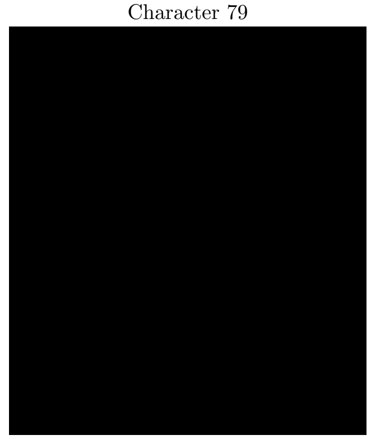
Character 77



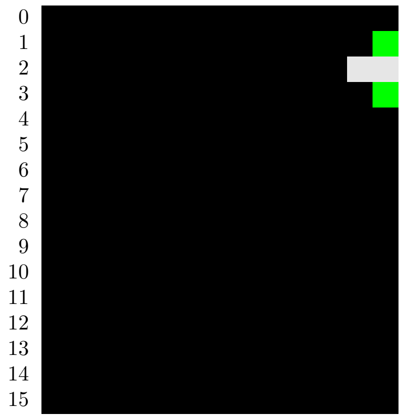
Character 78



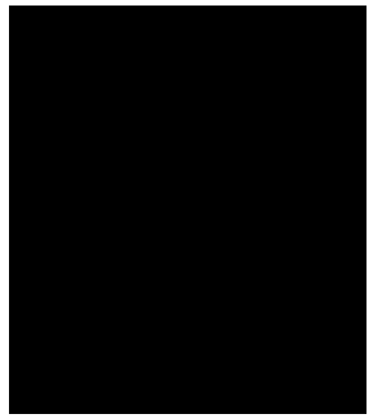
Character 79



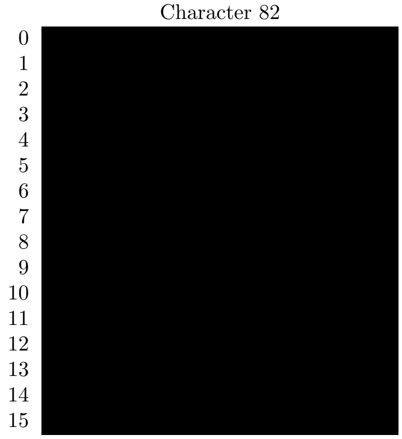
Character 80



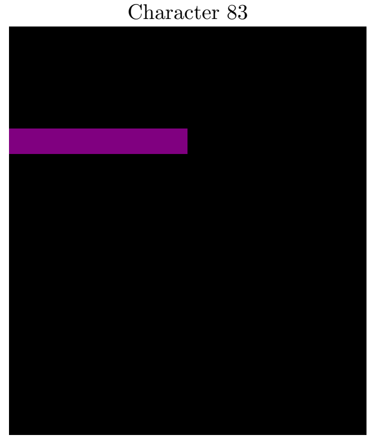
Character 81



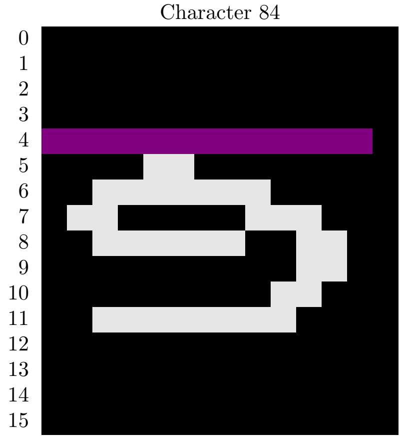
Character 82



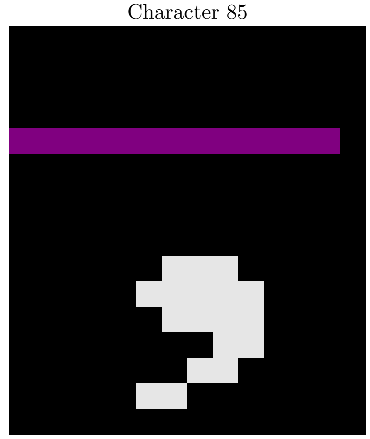
Character 83



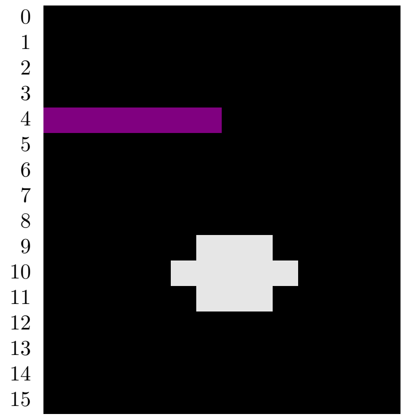
Character 84



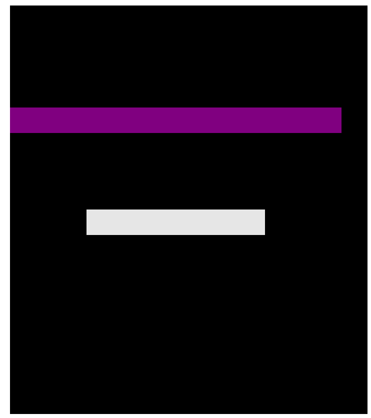
Character 85



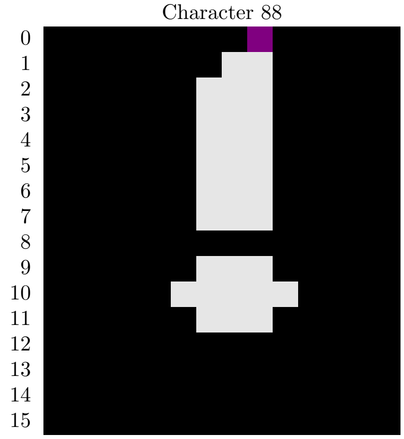
Character 86



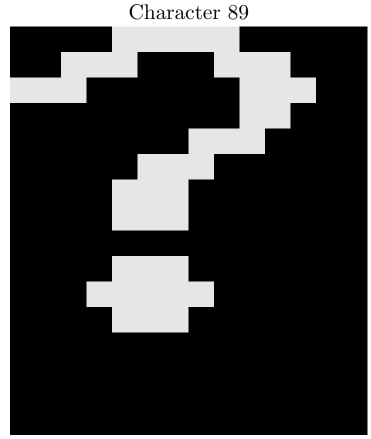
Character 87



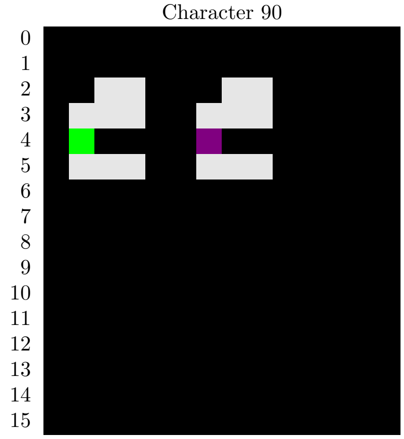
Character 88



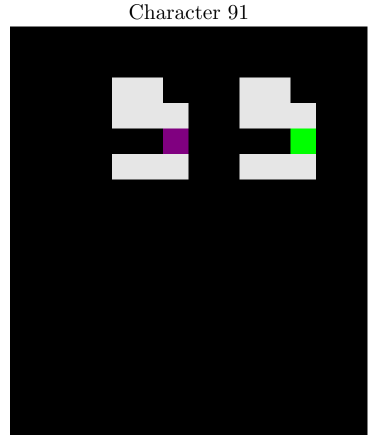
Character 89



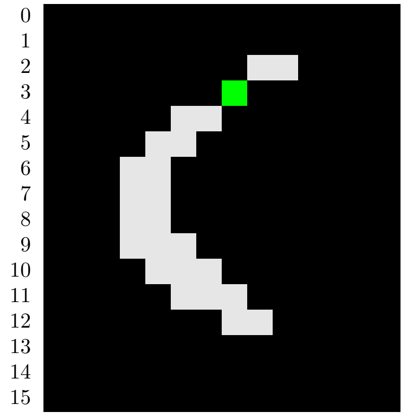
Character 90



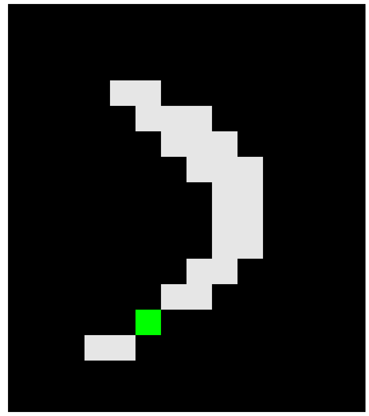
Character 91



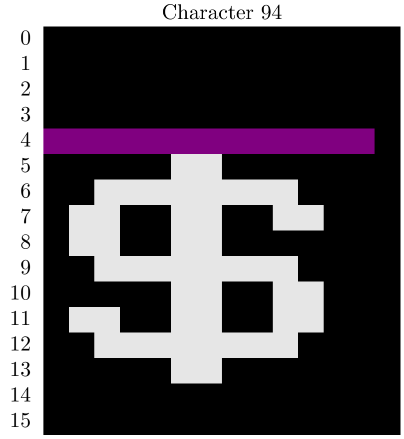
Character 92



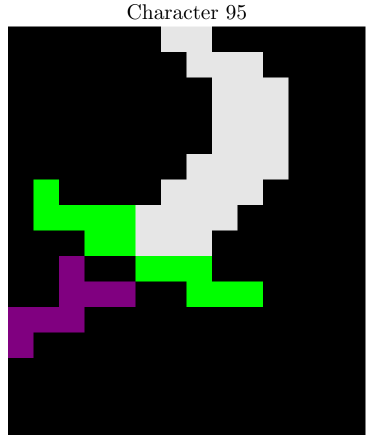
Character 93



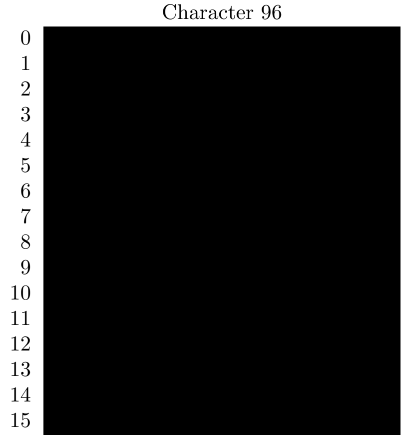
Character 94



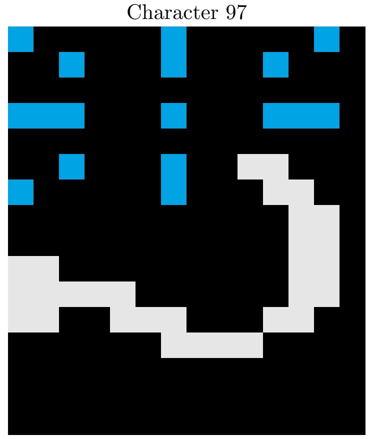
Character 95



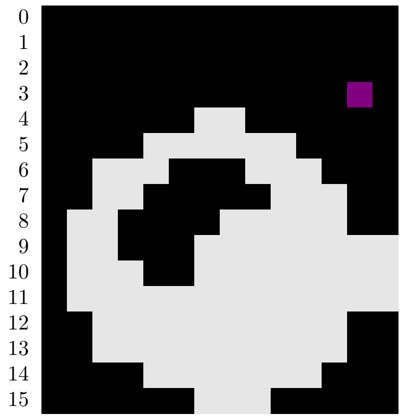
Character 96



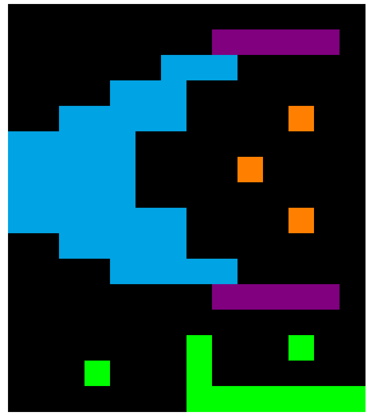
Character 97



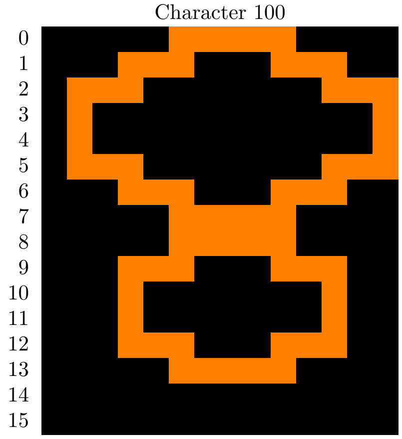
Character 98



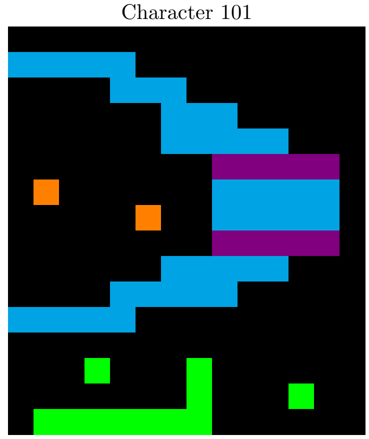
Character 99



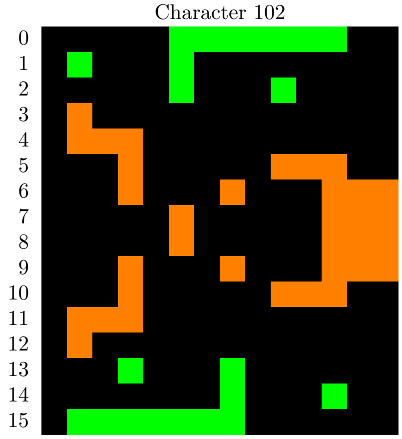
Character 100



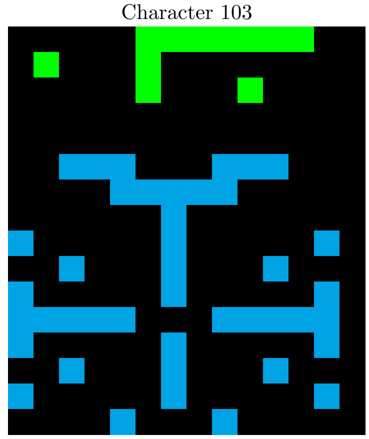
Character 101



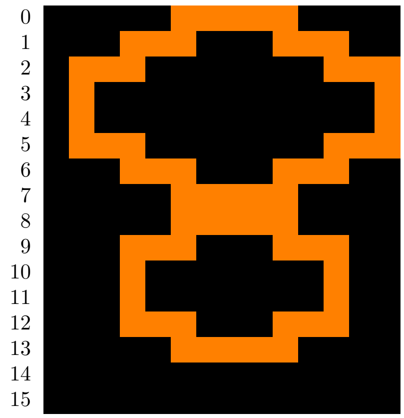
Character 102



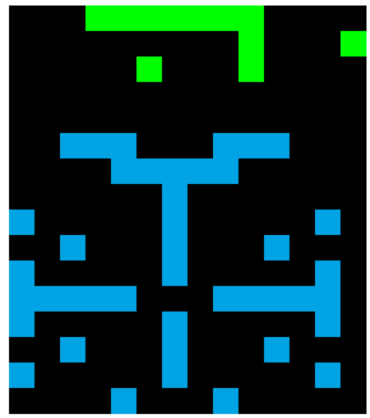
Character 103



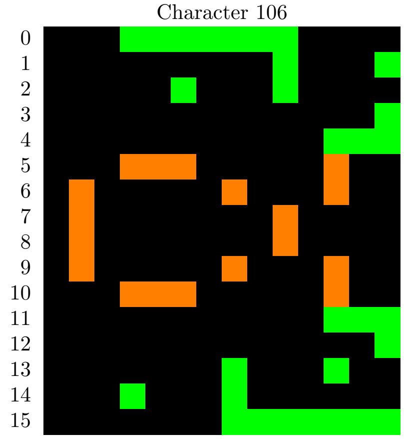
Character 104



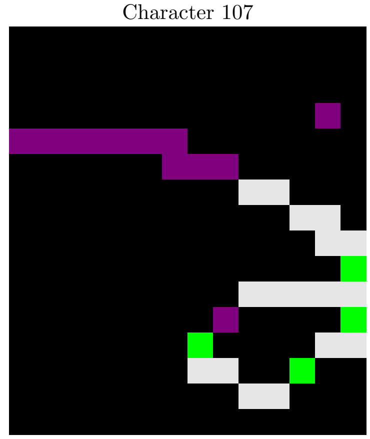
Character 105



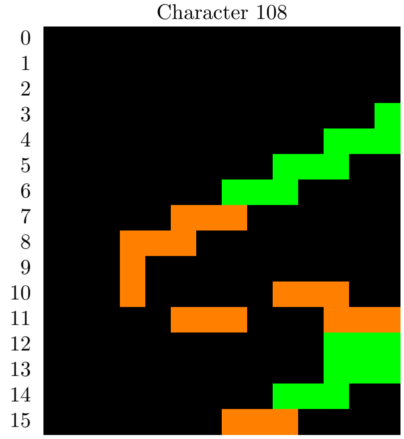
Character 106



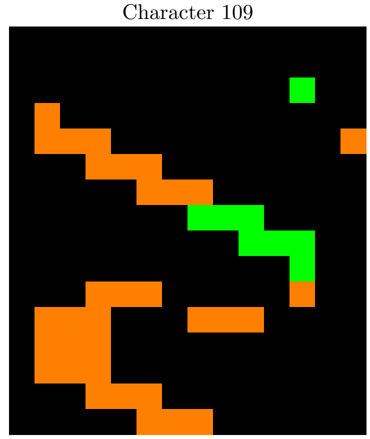
Character 107



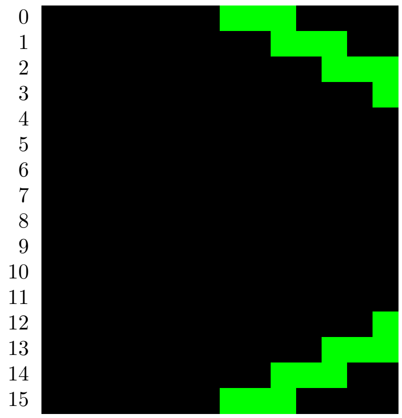
Character 108



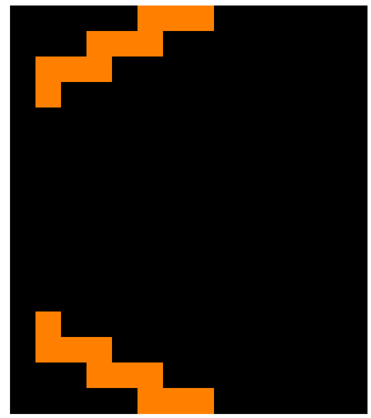
Character 109



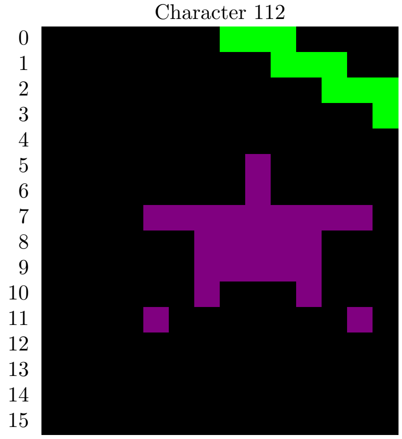
Character 110



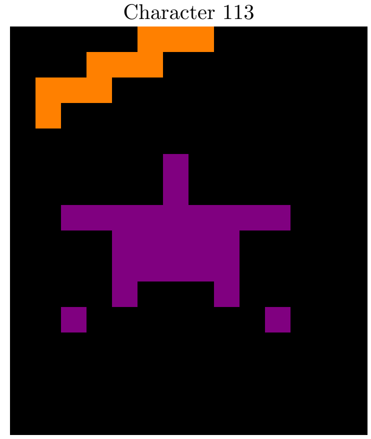
Character 111



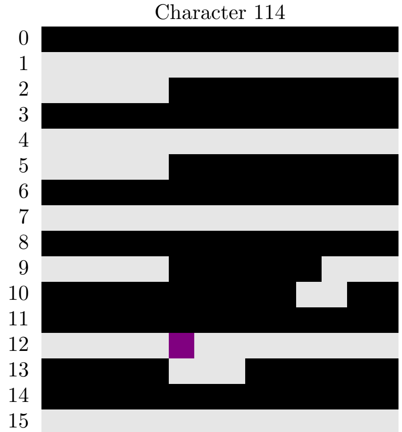
Character 112



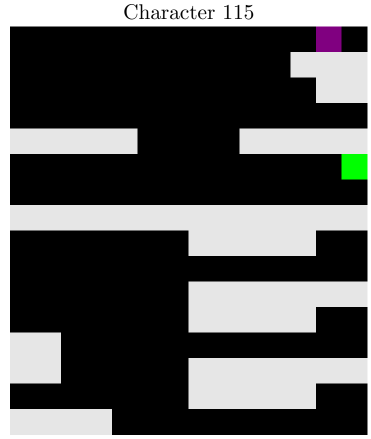
Character 113



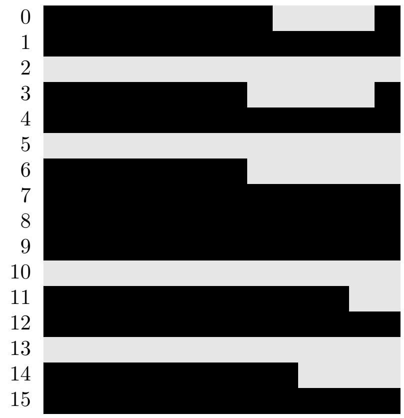
Character 114



Character 115

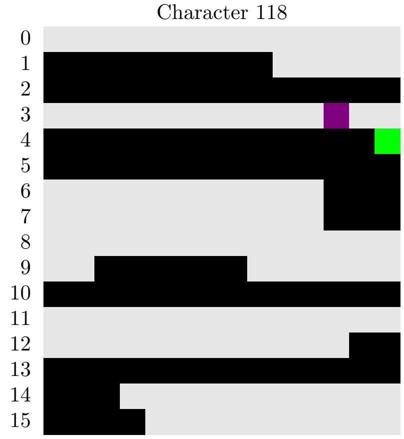


Character 116

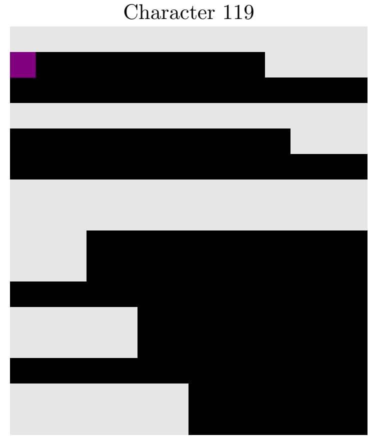


Character 117

Character 118



Character 119



Chapter 6

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 7

The main program

This is the Z-machine proper.

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

53a ⟨*main* 53a⟩≡ (231) 53b ▷

```
  main:
      SUBROUTINE

      CLD
      LDA      #$00
      LDX      #$80

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

Defines:
`main`, used in chunks 17, 23a, 56c, 58a, 67, 70, 222b, and 224b.

And we reset the 6502 stack pointer.

53b ⟨*main* 53a⟩+≡ (231) ◁ 53a 54a ▷

```
      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.

54a ⟨*main* 53a⟩+≡ (231) ◁53b 54b▷

```
.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 227.

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.

54b ⟨*main* 53a⟩+≡ (231) ◁54a 54c▷

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

Uses ZCODE_PAGE_VALID 227 and ZCODE_PAGE_VALID2 227.

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

54c ⟨*main* 53a⟩+≡ (231) ◁54b 54d▷

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

Uses STACK_COUNT 227, STOB 8b, STOW 7, ZCHAR_SCRATCH1 227, and Z_SP 227.

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.

54d ⟨*main* 53a⟩+≡ (231) ◁54c 55a▷

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

Uses NEXT_PAGE_TABLE 227, PAGE_H_TABLE 227, PAGE_L_TABLE 227, PREV_PAGE_TABLE 227, and STOW 7.

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.

55a ⟨*main* 53a⟩+≡ (231) ◁54d 55b▷

```
    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 227, PAGE_H_TABLE 227, PAGE_L_TABLE 227, and PREV_PAGE_TABLE 227.

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).

55b ⟨*main* 53a⟩+≡ (231) ◁55a 55c▷

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

Uses FIRST_Z_PAGE 227, LAST_Z_PAGE 227, and STOB 8b.

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.

55c ⟨*main* 53a⟩+≡ (231) ◁55b 56a▷

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

Uses STOW 7.

Then we clear the screen.

56a ⟨*main* 53a⟩+≡ (231) ◁55c 56c▷

```
        JSR      do_reset_window
```

Uses `do_reset_window` 56b.

56b ⟨*Do reset window* 56b⟩≡ (230b)

```
  do_reset_window:
        JSR      reset_window
        RTS
```

Defines:
`do_reset_window`, used in chunk 56a.
Uses `reset_window` 75b.

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.

56c ⟨*main* 53a⟩+≡ (231) ◁56a 57▷

```
  .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` 8c, `SCRATCH1` 227, `SCRATCH2` 227, `STOW` 7, `main` 53a, and `read_from_sector` 131.

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.

57 ⟨*main* 53a⟩+≡ (231) ◁56c 58a▷

```
.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 227.

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

58a ⟨*main* 53a⟩+≡ (231) ◁57 58b▷

```
    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 227, SCRATCH1 227, SCRATCH2 227, main 53a, and read_from_sector 131.

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.

58b ⟨*main* 53a⟩+≡ (231) ◁58a 59c▷

```
.check_debug_flag:
    LDY     #HEADER_FLAGS1
    LDA     (Z_HEADER_ADDR),Y
    AND     #$01
    EOR     #$01
    BEQ     .brk
```

Uses brk 59b.

59a ⟨*die* 59a⟩≡ (62b)
```
  .brk:
      JSR      brk
```
Uses brk 59b.

59b ⟨*brk* 59b⟩≡ (231)
```
  brk:
      BRK
```
Defines:
brk, used in chunks 58b, 59a, 61, 63, 64, 181, 200b, 215, and 220.

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.

59c ⟨*main* 53a⟩+≡ (231) ◁58b 60▷
```
  .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 8b and Z_PC 227.

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.

60 ⟨*main* 53a⟩+≡ (231) ◁59c 61▷

```
  .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 227 and Z_ABBREV_TABLE 227.

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.

61 ⟨*main* 53a⟩+≡ (231) ◁60 62b▷

```
    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 227, LAST_Z_PAGE 227, NEXT_PAGE_TABLE 227, NUM_IMAGE_PAGES 227, STOB 8b, and brk 59b.

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.

62a ⟨*Locate last RAM page* 62a⟩≡ (231)

```
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 8b and SCRATCH2 227.

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

62b ⟨*main* 53a⟩+≡ (231) ◁61

```
    JMP       do_instruction

⟨die 59a⟩
```

Uses do_instruction 136.

Chapter 8

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.

63 ⟨*Push* 63⟩≡ (231)

```
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 147, 148, 150–52, and 192b.
Uses SCRATCH2 227, STACK_COUNT 227, SUBB 13a, Z_SP 227, and brk 59b.

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.

64 ⟨*Pop* 64⟩≡ (231)

```
    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 145, 148, 154, 191b, 192a, and 206a.
Uses INCW 10a, SCRATCH2 227, STACK_COUNT 227, Z_SP 227, and brk 59b.

Chapter 9

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.

65 ⟨*Get next code byte* 65⟩≡ (231) 66▷

```
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 66, 136, 137a, 144, 145, 147, 150c, 151, 184, and 185.
Uses INCW 10a, ZCODE_PAGE_ADDR 227, ZCODE_PAGE_VALID 227, and Z_PC 227.

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.

66 ⟨*Get next code byte* 65⟩+≡ (231) ◁65 67▷

```
  .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 8c, NUM_IMAGE_PAGES 227, PAGE_TABLE_INDEX 227, SCRATCH2 227, STOB 8b, ZCODE_PAGE_ADDR 227, ZCODE_PAGE_VALID 227, Z_PC 227, find_index_of_page_table 68, get_next_code_byte 65, and set_page_first 69.

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.

67 ⟨*Get next code byte* 65⟩+≡ (231) ◁66

```
.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 227, MOVW 8c, PAGE_H_TABLE 227, PAGE_L_TABLE 227, PAGE_TABLE_INDEX 227, PAGE_TABLE_INDEX2 227, SCRATCH1 227, SCRATCH2 227, STOB 8b, ZCODE_PAGE_VALID2 227, Z_PC 227, good_read 242, main 53a, read_from_sector 131, and set_page_first 69.

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.

68 ⟨*Find index of page table* 68⟩≡ (231)

```
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 66 and 70.
Uses LAST_Z_PAGE 227, PAGE_H_TABLE 227, PAGE_L_TABLE 227, and SCRATCH2 227.

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.

69 ⟨*Set page first 69*⟩≡ (231)

```
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 66, 67, and 70.

Uses FIRST_Z_PAGE 227, LAST_Z_PAGE 227, MOVB 8b, NEXT_PAGE_TABLE 227, PREV_PAGE_TABLE 227, and SCRATCH2 227.

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.

70 ⟨*Get next code byte 2* 70⟩≡ (231)

```
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 72a and 189a.

Uses HIGH_MEM_ADDR 227, MOVW 8c, NUM_IMAGE_PAGES 227, PAGE_H_TABLE 227, PAGE_L_TABLE 227, PAGE_TABLE_INDEX 227, PAGE_TABLE_INDEX2 227, SCRATCH1 227, SCRATCH2 227, STOB 8b, ZCODE_PAGE_ADDR2 227, ZCODE_PAGE_VALID 227, ZCODE_PAGE_VALID2 227, Z_PC2_H 227, Z_PC2_HH 227, Z_PC2_L 227, find_index_of_page_table 68, good_read 242, main 53a, read_from_sector 131, and set_page_first 69.

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`.

72a ⟨*Get next code word* 72a⟩≡ (231)

```
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 85 and 188b.
Uses `SCRATCH2` 227 and `get_next_code_byte2` 70.

The `load_address` routine copies `SCRATCH2` to `Z_PC2`.

72b ⟨*Load address* 72b⟩≡ (231)

```
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 160b, 188b, 189a, and 208b.
Uses `MOVB` 8b, `SCRATCH2` 227, `STOB` 8b, `Z_PC2_H` 227, `Z_PC2_HH` 227, and `Z_PC2_L` 227.

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

73 ⟨*Load packed address* 73⟩≡ (231)
```
  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 89 and 209c.

Uses SCRATCH2 227, STOB 8b, ZCODE_PAGE_VALID2 227, Z_PC2_H 227, Z_PC2_HH 227, and Z_PC2_L 227.

Chapter 10

I/O

10.1 Strings and output

10.1.1 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.

74 ⟨*Output string to console* 74⟩≡ (231)

```
cout_string:
    SUBROUTINE

    LDY     #$00

.loop:
```

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

Defines:
cout_string, used in chunks 79, 93, and 168.
Uses COUT1 226 and SCRATCH2 227.

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.

75a ⟨*Home* 75a⟩≡ (231)

```
home:
    SUBROUTINE

    JSR     HOME
    MOVB    WNDTOP, CURR_LINE
    RTS
```

Defines:
home, used in chunks 75b and 166.
Uses CURR_LINE 227, HOME 226, MOVB 8b, and WNDTOP 226.

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.

75b ⟨*Reset window* 75b⟩≡ (231)

```
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 56b.
Uses INVFLG 226, PROMPT 226, STOB 8b, WNDBTM 226, WNDLFT 226, WNDTOP 226, WNDWDTH 226, and home 75a.

10.1.2 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.

76 ⟨*Dump buffer to screen* 76⟩≡ (231)

```
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 78 and 93.
Uses BUFF_AREA 227, BUFF_END 227, and COUT1 226.

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.

77 ⟨*Dump buffer to printer* 77⟩≡ (231)

```
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 78 and 95.
`printer_card_initialized_flag`, never used.
Uses `BUFF_AREA` 227, `BUFF_END` 227, `COUT` 226, `CSW` 226, `MOVW` 8c, `PRINTER_CSW` 227, `PSHW` 9a, `PULW` 9c, and `STOB` 8b.

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`.

78 ⟨*Dump buffer line* 78⟩≡ (231)

```
  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 80a, 93, 95, 168, 170a, and 171.
Uses `HEADER_FLAGS2` 229a, `dump_buffer_to_printer` 77, and `dump_buffer_to_screen` 76.

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.

79 ⟨*Dump buffer with more* 79⟩≡ (231) 80a ▷

```
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 81, 82b, 166, 168, 170, 171, 222b, and 223.

Uses `CH` 226, `CLREOL` 226, `CURR_LINE` 227, `INVFLG` 226, `MOVB` 8b, `RDKEY` 226, `SCRATCH2` 227, `STOB` 8b, `STOW` 7, `WNDBTM` 226, `WNDTOP` 226, and `cout_string` 74.

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.

80a ⟨*Dump buffer with more 79*⟩+≡ (231) ◁79 80b▷

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

    .skip_newline:
```

Uses `BUFF_END` 227, `COUT1` 226, `WNDWDTH` 226, and `dump_buffer_line` 78.

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.

80b ⟨*Dump buffer with more 79*⟩+≡ (231) ◁80a 80c▷

```
        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` 226, `CSW` 226, `HEADER_FLAGS2` 229a, `MOVW` 8c, `PRINTER_CSW` 227, `PSHW` 9a, and `PULW` 9c.

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

80c ⟨*Dump buffer with more 79*⟩+≡ (231) ◁80b

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

Uses `buffer_char_set_buffer_end` 81.

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.

81 ⟨*Buffer a character* 81⟩≡ (231) 82a ▷

```
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 83b, 90a, 91c, 93, 127, 128, 167a, 169, 205b, 207b, and 208c.
buffer_char_set_buffer_end, used in chunk 80c.

Uses BUFF_AREA 227, BUFF_END 227, WNDWDTH 226, and dump_buffer_with_more 79.

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.

82a ⟨*Buffer a character* 81⟩+≡ (231) ◁81 82b▷

```
    LDA      #$A0  ; normal space

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

.endloop:
```

Uses BUFF_AREA 227 and WNDWDTH 226.

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.

82b ⟨*Buffer a character* 81⟩+≡ (231) ◁82a 83a▷

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

Uses BUFF_END 227, BUFF_LINE_LEN 227, and dump_buffer_with_more 79.

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.

83a ⟨*Buffer a character* 81⟩+≡ (231) ◁82b

```
  .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 227, BUFF_END 227, BUFF_LINE_LEN 227, and WNDWDTH 226.

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.

83b ⟨*Print ASCII string* 83b⟩≡ (230b)

```
  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 166, 168, 170a, 171, and 223.
Uses SCRATCH2 227, SCRATCH3 227, and buffer_char 81.

10.1.3 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.

84 ⟨*Get alphabet* 84⟩≡ (231)

```
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 87a and 88.
Uses LOCKED_ALPHABET 227 and SHIFT_ALPHABET 227.

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.

85 ⟨*Get next zchar* 85⟩≡ (231)

```
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 87a, 89, and 92a.
Uses MOVW 8c, SCRATCH2 227, STOB 8b, ZCHARS_H 227, ZCHARS_L 227, ZDECOMPRESS_STATE 227, and get_next_code_word 72a.

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.

86 ⟨*Print zstring* 86⟩≡ (231) 87a ▷

```
print_zstring:
    SUBROUTINE

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

Defines:
  print_zstring, used in chunks 89, 92b, 160b, and 182b.
Uses LOCKED_ALPHABET 227, SHIFT_ALPHABET 227, STOB 8b, and ZDECOMPRESS_STATE 227.

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.

87a ⟨*Print zstring* 86⟩+≡ (231) ◁86

```
.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 90a⟩
```

Uses `SCRATCH3` 227, `get_alphabet` 84, and `get_next_zchar` 85.

87b ⟨*Printing a space* 87b⟩≡ (231)

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

Defines:
`.space`, never used.

88 ⟨*Shifting alphabets* 88⟩≡ (231)

```
.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` 126, `LOCKED_ALPHABET` 227, `SCRATCH3` 227, `SHIFT_ALPHABET` 227, and `get_alphabet` 84.

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.

89 ⟨*Printing an abbreviation* 89⟩≡ (231)

```
.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` 227, `SCRATCH2` 227, `SHIFT_ALPHABET` 227, `STOB` 8b, `ZCHARS_H` 227, `ZCHARS_L` 227, `ZCODE_PAGE_VALID2` 227, `ZDECOMPRESS_STATE` 227, `Z_ABBREV_TABLE` 227, `Z_PC2_H` 227, `Z_PC2_HH` 227, `Z_PC2_L` 227, `get_next_zchar` 85, `load_packed_address` 73, and `print_zstring` 86.

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`.

90a ⟨*Print the zchar* 90a⟩≡ (87a) 90b▷

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

.add_ascii_offset:
    CLC
    ADC     SCRATCH3

.printchar:
    JSR     buffer_char
    JMP     .loop
```

Uses `SCRATCH3` 227 and `buffer_char` 81.

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

90b ⟨*Print the zchar* 90a⟩+≡ (87a) ◁90a 91b▷

```
.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.

91a ⟨*A2 table* 91a⟩≡ (231)

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

Defines:
a2_table, used in chunks 91b and 111.

91b ⟨*Print the zchar* 90a⟩+≡ (87a) ◁90b

```
  .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 227 and a2_table 91a.

91c ⟨*Printing a CRLF* 91c⟩≡ (231)

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

Defines:
.crlf, never used.
Uses buffer_char 81.

92a ⟨*Printing a 10-bit ZSCII character* 92a⟩≡ (231)
```
  .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` 227 and `get_next_zchar` 85.

`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`.

92b ⟨*Printing a string literal* 92b⟩≡ (231)
```
  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` 8b, `MOVW` 8c, `STOB` 8b, `ZCODE_PAGE_ADDR` 227, `ZCODE_PAGE_ADDR2` 227, `ZCODE_PAGE_VALID` 227, `ZCODE_PAGE_VALID2` 227, `Z_PC` 227, `Z_PC2_H` 227, `Z_PC2_HH` 227, `Z_PC2_L` 227, and `print_zstring` 86.

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.

93 ⟨*Print status line* 93⟩≡ (231)

```
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 97.
  `sScore`, never used.
Uses `CH` 226, `CLREOL` 226, `CV` 226, `INVFLG` 226, `SCRATCH2` 227, `STOB` 8b, `STOW` 7, `VAR_CURR_ROOM` 229b, `VAR_MAX_SCORE` 229b, `VAR_SCORE` 229b, `VTAB` 226, `buffer_char` 81, `cout_string` 74, `dump_buffer_line` 78, `dump_buffer_to_screen` 76, `print_number` 127, `print_obj_in_A` 160b, and `var_get` 146.

10.1.4 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.

95 ⟨*Read line* 95⟩≡ (231)

```
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 97.

Uses `BUFF_AREA` 227, `BUFF_END` 227, `CURR_LINE` 227, `GETLN1` 226, `HEADER_FLAGS2` 229a, `MOVB` 8b, `OPERAND0` 227, `STOB` 8b, `WNDTOP` 226, `dump_buffer_line` 78, and `dump_buffer_to_printer` 77.

10.1.5 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.

97 ⟨*Instruction sread* 97⟩≡ (231) 98a ▷

```
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 133.

Uses `ADDW` 12a, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH3` 227, `STOB` 8b, `print_status_line` 93, and `read_line` 95.

Loop:

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

98a ⟨*Instruction sread* 97⟩+≡ (231) ◁97 98b▷
```
  .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 227 and do_instruction 136.

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

98b ⟨*Instruction sread* 97⟩+≡ (231) ◁98a 98c▷
```
  .not_end1:
      LDA      SCRATCH3+1     ; if input_count == char_count == 0 do_instruction
      ORA      SCRATCH3
      BNE      .not_end2
      JMP      do_instruction
```
Uses SCRATCH3 227 and do_instruction 136.

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

98c ⟨*Instruction sread* 97⟩+≡ (231) ◁98b 99a▷
```
  .not_end2:
      LDA      SCRATCH3       ; if char_count != 6 .not_min_compress_size
      CMP      #$06
      BNE      .not_min_compress_size
      JSR      skip_separators
```
Uses SCRATCH3 227 and skip_separators 102.

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

99a ⟨*Instruction sread* 97⟩+≡ (231) ◁98c 99b▷

```
  .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 227 and ZCHAR_SCRATCH1 227.

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.

99b ⟨*Instruction sread* 97⟩+≡ (231) ◁99a 100a▷

```
      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 227, OPERAND1 227, SCRATCH3 227, is_dict_separator 103, and is_std_separator 103.

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.

100a ⟨*Instruction sread* 97⟩+≡ (231) ◁99b 100b▷

```
  .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 227, SCRATCH3 227, ZCHAR_SCRATCH1 227, and is_separator 103.

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.

100b ⟨*Instruction sread* 97⟩+≡ (231) ◁100a 101a▷

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

Uses SCRATCH3 227, ZCHAR_SCRATCH1 227, and is_dict_separator 103.

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.

101a ⟨*Instruction sread* 97⟩+≡ (231) ◁100b 101b▷

```
.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 227 and SCRATCH3 227.

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.

101b ⟨*Instruction sread* 97⟩+≡ (231) ◁101a

```
        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 227, SCRATCH1 227, SCRATCH3 227, STOB 8b, ascii_to_zchar 104, and match_dictionary_word 114.

Separators

102 ⟨*Skip separators 102*⟩≡ (231)

```
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 98c.

Uses `OPERAND0` 227, `SCRATCH3` 227, and `is_separator` 103.

103 ⟨*Separator checks* 103⟩≡ (231)
```
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 99b and 100b.
is_separator, used in chunks 100a and 102.
is_std_separator, used in chunk 99b.
separator_found, never used.
separator_not_found, never used.
Uses SCRATCH2 227 and get_dictionary_addr 113.

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.

104 ⟨*ASCII to Zchar* 104⟩≡ (231) 105a ▷

```
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 101b.
Uses LOCKED_ALPHABET 227, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, STOB 8b, and ZCHAR_SCRATCH2 227.

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.

105a ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁104 105b▷

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

Uses SCRATCH2 227, SCRATCH3 227, and ZCHAR_SCRATCH1 227.

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.

105b ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁105a 106b▷

```
  .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 227, SCRATCH1 227, SCRATCH3 227, and get_alphabet_for_char 106a.

106a ⟨*Get alphabet for char* 106a⟩≡ (231)

```
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 105b, 106b, and 109b.

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.

106b ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁105b 107a▷

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

Uses SCRATCH1 227, SCRATCH2 227, ZCHAR_SCRATCH1 227, and get_alphabet_for_char 106a.

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.

107a ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁106b 107b▷

```
        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_3 126, LOCKED_ALPHABET 227, MOVB 8b, and SCRATCH1 227.

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

107b ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁107a 107c▷

```
        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 227 and ZCHAR_SCRATCH2 227.

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.

107c ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁107b 109a▷

```
        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 227, SCRATCH3 227, and z_compress 108.

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

108 ⟨*Z compress* 108⟩≡ (231)

```
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 107c, 109a, 110a, and 112.
Uses ZCHAR_SCRATCH2 227.

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.

109a ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁107c 109b▷

```
  .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_3 126, LOCKED_ALPHABET 227, SCRATCH1 227, SCRATCH3 227, ZCHAR_SCRATCH2 227, and z_compress 108.

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

109b ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁109a 110a▷

```
  .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` 227, `SCRATCH3` 227, and `get_alphabet_for_char` 106a.

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.

110a ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁109b 110b▷

```
  .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` 227, `SCRATCH3` 227, `ZCHAR_SCRATCH2` 227, and `z_compress` 108.

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.

110b ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁110a 110c▷

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

Uses `SCRATCH3` 227.

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.

110c ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁110b 112▷

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

Uses `SCRATCH3` 227 and `search_nonalpha_table` 111.

111 ⟨*Search nonalpha table* 111⟩≡ (231)

```
  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 110c.
Uses `a2_table` 91a.

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.

112 ⟨*ASCII to Zchar* 104⟩+≡ (231) ◁110c

```
    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 227, SCRATCH3 227, ZCHAR_SCRATCH2 227, and z_compress 108.

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`.

113 ⟨*Get dictionary address* 113⟩≡ (231)

```
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 103 and 114.

Uses `ADDW` 12a, `HEADER_DICT_ADDR` 229a, and `SCRATCH2` 227.

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.

114 ⟨*Match dictionary word* 114⟩≡ (231) 115a ▷

```
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 101b.
Uses ADDA 10b, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, and get_dictionary_addr 113.

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.

115a ⟨*Match dictionary word* 114⟩+≡ (231) ◁114 115b▷

```
.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 11c, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, and ZCHAR_SCRATCH2 227.

115b ⟨*Match dictionary word* 114⟩+≡ (231) ◁115a 116▷

```
.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 11c, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, and SUBB2 13b.

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.

116 ⟨*Match dictionary word* 114⟩+≡ (231) ◁115b 117a▷

```
.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 11c, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, SUBB 13a, and ZCHAR_SCRATCH2 227.

If the search failed, we return 0 in SCRATCH1.

117a ⟨*Match dictionary word* 114⟩+≡ (231) ◁116 117b▷

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

Uses SCRATCH1 227.

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

117b ⟨*Match dictionary word* 114⟩+≡ (231) ◁117a

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

Uses SCRATCH1 227, SCRATCH2 227, and SUBW 14a.

Chapter 11

Arithmetic routines

11.1 Negation and sign manipulation

`negate` negates the word in `SCRATCH2`.

118 ⟨*negate* 118⟩≡ (231)

```
  negate:
      SUBROUTINE

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

Defines:
  `negate`, used in chunks 119a, 120, and 128.
Uses `SCRATCH2` 227 and `SUBWL` 14b.

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.

119a ⟨*Flip sign* 119a⟩≡ (231)

```
flip_sign:
    SUBROUTINE

    ORA      #$00
    BMI      .do_negate
    RTS

.do_negate:
    INC      SIGN_BIT
    JMP      negate
```

Defines:
flip_sign, used in chunk 119b.
Uses negate 118.

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.

119b ⟨*Check sign* 119b⟩≡ (231)

```
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 194–96.
Uses SCRATCH1 227, SCRATCH2 227, STOB 8b, and flip_sign 119a.

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.

120 ⟨*Set sign* 120⟩≡ (231)
```
set_sign:
    SUBROUTINE

    LDA     SIGN_BIT
    AND     #$01
    BNE     negate
    RTS
```
Defines:
set_sign, used in chunk 196.
Uses negate 118.

11.2 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.

121 ⟨*mulu16* 121⟩≡ (231)

```
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 196.

Uses `ADDWC` 12b, `MOVW` 8c, `PSHW` 9a, `PULW` 9c, `RORW` 15b, `SCRATCH1` 227, `SCRATCH2` 227, `SCRATCH3` 227, and `STOW` 7.

11.3 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:

123 ⟨*trace of divu16* 123⟩≡

```
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.

124 ⟨*divu16* 124⟩≡ (231)

```
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 127, 194, 195, and 197a.

Uses MOVW 8c, PSHW 9a, PULW 9c, ROLW 15a, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, and STOW 7.

11.4 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.

125a ⟨*cmpu16* 125a⟩≡ (231)

```
cmpu16:
    SUBROUTINE

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

Defines:
  `cmpu16`, used in chunks 125b and 204a.
Uses `SCRATCH1` 227 and `SCRATCH2` 227.

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

125b ⟨*cmp16* 125b⟩≡ (231)

```
cmp16:
    SUBROUTINE

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

Defines:
  `cmp16`, used in chunks 200a, 202a, and 203a.
Uses `SCRATCH1` 227, `SCRATCH2` 227, and `cmpu16` 125a.

11.5 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.

126 ⟨*A mod 3* 126⟩≡ (231)

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

.end:
    RTS
```

Defines:
A_mod_3, used in chunks 88, 107a, and 109a.

11.6 Printing numbers

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

127 ⟨*Print number* 127⟩≡ (231)

```
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 93 and 209a.

Uses `SCRATCH1` 227, `SCRATCH2` 227, `SCRATCH3` 227, `STOB` 8b, `STOW` 7, `buffer_char` 81, `divu16` 124, and `print_negative_num` 128.

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.

128 ⟨*Print negative number* 128⟩≡ (231)

```
print_negative_num:
    SUBROUTINE

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

Defines:
  `print_negative_num`, used in chunk 127.

Uses `buffer_char` 81 and `negate` 118.

Chapter 12

Disk routines

129 ⟨*iob struct* 129⟩≡ (230b)

```
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 132.
`iob`, used in chunks 130, 169, and 171.

`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.

130 ⟨*Do RWTS on sector* 130⟩≡ (230b)

```
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 131 and 132.

Uses `MOVW` 8c, `RWTS` 227, `SCRATCH1` 227, `SCRATCH2` 227, `SECTORS_PER_TRACK` 227, `STOB` 8b, and `iob` 129.

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`.

131 ⟨*Reading sectors* 131⟩≡ (230b)

```
  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 177b.
`read_from_sector`, used in chunks 56c, 58a, 67, and 70.
`read_next_sector`, used in chunks 175c and 177a.

Uses `BUFF_AREA` 227, `INCW` 10a, `SCRATCH1` 227, `SCRATCH2` 227, `STOW` 7, and `do_rwts_on_sector` 130.

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.

132 ⟨*Writing sectors* 132⟩≡ (230b)

```
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 174b.
  write_next_sector, used in chunks 173b and 174a.

Uses BUFF_AREA 227, INCW 10a, PSHW 9a, PULW 9c, SCRATCH1 227, SCRATCH2 227, STOW 7, STOW2 8a, dct 129, and do_rwts_on_sector 130.

Chapter 13

The instruction dispatcher

13.1 Executing an instruction

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

133 ⟨*Instruction tables* 133⟩≡ (230a)

```
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 137b.
routines_table_1op, used in chunk 139b.
routines_table_2op, used in chunk 141c.

routines_table_var, used in chunk 143.
Uses illegal_opcode 181, instr_add 193b, instr_and 198a, instr_call 149, instr_clear_attr 210, instr_dec 193a, instr_dec_chk 199b, instr_div 194, instr_get_next_prop 212, instr_get_parent 213a, instr_get_prop 213b, instr_get_prop_addr 216, instr_get_prop_len 217, instr_get_sibling 218, instr_inc 192c, instr_inc_chk 200a, instr_insert_obj 219, instr_je 200b, instr_jg 202a, instr_jin 202b, instr_jl 203a, instr_jump 205a, instr_jz 203b, instr_load 188a, instr_loadb 189a, instr_loadw 188b, instr_mod 195, instr_mul 196, instr_new_line 207b, instr_nop 222a, instr_not 198b, instr_or 199a, instr_pop 191b, instr_print 208a, instr_print_addr 208b, instr_print_char 208c, instr_print_num 209a, instr_print_obj 209b, instr_print_paddr 209c, instr_print_ret 205b, instr_pull 192a, instr_push 192b, instr_put_prop 220, instr_quit 223, instr_random 197a, instr_remove_obj 221a, instr_restart 222b, instr_restore 175b, instr_ret 153, instr_ret_popped 206a, instr_rfalse 206b, instr_rtrue 207a, instr_save 172a, instr_set_attr 221b, instr_sread 97, instr_store 189b, instr_storeb 191a, instr_storew 190, instr_sub 197c, instr_test 204a, and instr_test_attr 204b.

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.

135 ⟨*Execute instruction* 135⟩≡ (230a)

```
.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 227, SCRATCH1 227, and SCRATCH2 227.

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.

13.2 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`.

136 ⟨*Do instruction* 136⟩≡ (230a) 137a ▷

```
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 62b, 98, 152b, 182, 184b, 187, 189–93, 207–10, and 219–22.

Uses `CURR_OPCODE` 227, `MOVB` 8b, `MOVW` 8c, `OPERAND_COUNT` 227, `STOB` 8b, `TMP_Z_PC` 227, `Z_PC` 227, and `get_next_code_byte` 65.

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

13.3 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.

137a ⟨*Do instruction* 136⟩+≡ (230a) ◁136

```
    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 142⟩
```

Uses get_next_code_byte 65.

13.3.1 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.

137b ⟨*Handle 0op instructions* 137b⟩≡ (230a)

```
.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 227, STOW 7, illegal_opcode 181, and routines_table_0op 133.

13.3.2 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.

138a ⟨*Handle 1op instructions* 138a⟩≡ (230a) 138b ▷

```
.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` 144b.

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

138b ⟨*Handle 1op instructions* 138a⟩+≡ (230a) ◁ 138a 138c ▷

```
.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` 144a.

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

138c ⟨*Handle 1op instructions* 138a⟩+≡ (230a) ◁ 138b 138d ▷

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

Uses `get_var_content` 145.

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

138d ⟨*Handle 1op instructions* 138a⟩+≡ (230a) ◁ 138c 139a ▷

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

Uses `MOVW` 8c, `OPERAND0` 227, `OPERAND_COUNT` 227, `SCRATCH2` 227, and `STOB` 8b.

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.

139a ⟨*Handle 1op instructions* 138a⟩+≡ (230a) ◁138d 139b▷

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

Uses CURR_OPCODE 227 and illegal_opcode 181.

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

139b ⟨*Handle 1op instructions* 138a⟩+≡ (230a) ◁139a

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

Uses SCRATCH2 227, STOW 7, and routines_table_1op 133.

13.3.3 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`.

140a ⟨*Handle 2op instructions* 140a⟩≡ (230a) 140b ▷

```
.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` 8c, `OPERAND0` 227, `SCRATCH2` 227, `get_const_byte` 144a, and `get_var_content` 145.

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`.

140b ⟨*Handle 2op instructions* 140a⟩+≡ (230a) ◁ 140a 141a ▷

```
    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` 227, `MOVW` 8c, `OPERAND1` 227, `SCRATCH2` 227, `get_const_byte` 144a, and `get_var_content` 145.

OPERAND_COUNT is set to 2.

141a ⟨*Handle 2op instructions* 140a⟩+≡ (230a) ◁140b 141b▷

```
    STOB     #$02, OPERAND_COUNT
```

Uses OPERAND_COUNT 227 and STOB 8b.

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

141b ⟨*Handle 2op instructions* 140a⟩+≡ (230a) ◁141a 141c▷

```
    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 227 and illegal_opcode 181.

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

141c ⟨*Handle 2op instructions* 140a⟩+≡ (230a) ◁141b

```
.go_to_op2:
    PHA
    STOW     routines_table_2op, SCRATCH2
    PLA
    JMP      .opcode_table_jump
```

Uses SCRATCH2 227, STOW 7, and routines_table_2op 133.

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

13.3.4 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`.

142 ⟨*Handle varop instructions* 142⟩≡ (137a) 143▷

```
    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 227, OPERAND_COUNT 227, SCRATCH2 227, get_const_byte 144a, get_const_word 144b, and get_var_content 145.

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.

143 ⟨*Handle varop instructions* 142⟩+≡ (137a) ◁142

```
.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 227, SCRATCH2 227, STOW 7, illegal_opcode 181, and routines_table_var 133.

13.4 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`.

144a ⟨*Get const byte* 144a⟩≡ (230a)

```
get_const_byte:
    SUBROUTINE

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

Defines:
`get_const_byte`, used in chunks 138b, 140, and 142.
Uses `SCRATCH2` 227, `STOB` 8b, and `get_next_code_byte` 65.

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.

144b ⟨*Get const word* 144b⟩≡ (230a)

```
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 138a and 142.
Uses `SCRATCH2` 227 and `get_next_code_byte` 65.

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`.

145 ⟨*Get var content* 145⟩≡ (230a)

```
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 146.
`get_top_of_stack`, never used.
`get_var_content`, used in chunks 138c, 140, and 142.

Uses `ADDW` 12a, `GLOBAL_ZVARS_ADDR` 227, `LOCAL_ZVARS` 227, `SCRATCH1` 227, `SCRATCH2` 227, `Z_PC` 227, `get_next_code_byte` 65, and `pop` 64.

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

146 ⟨*Get var content in A* 146⟩≡ (230a)

```
var_get:
    SUBROUTINE

    ORA     #$00
    BEQ     pop_push
    JMP     get_nonstack_var
```

Defines:
`var_get`, used in chunks 93, 183, and 188a.

Uses `get_nonstack_var` 145 and `pop_push` 148.

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.

147 ⟨*Store var* 147⟩≡ (230a)

```
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 182a and 211.

Uses `ADDW` 12a, `GLOBAL_ZVARS_ADDR` 227, `LOCAL_ZVARS` 227, `PSHW` 9a, `PULW` 9c, `SCRATCH1` 227, `SCRATCH2` 227, `get_next_code_byte` 65, and `push` 63.

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.

148 ⟨*Store to var A* 148⟩≡ (230a)

```
    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 146.
`var_put`, used in chunks 183a and 189b.

Uses `PSHW` 9a, `PULW` 9c, `SCRATCH2` 227, `pop` 64, and `push` 63.

Chapter 14

Calls and returns

14.1 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.

149 ⟨*Instruction call* 149⟩≡ (231) 150a ▷

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

Defines:
`instr_call`, used in chunk 133.

Uses `OPERAND0` 227, `SCRATCH2` 227, `STOW` 7, and `store_and_next` 182a.

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.

150a ⟨*Instruction call* 149⟩+≡ (231) ◁149 150b▷

```
  .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 227, FRAME_Z_SP 227, MOVB 8b, MOVW 8c, SCRATCH2 227, STOB 8b, ZCODE_PAGE_VALID 227, Z_PC 227, and push 63.

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

150b ⟨*Instruction call* 149⟩+≡ (231) ◁150a 150c▷

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

Uses OPERAND0 227 and Z_PC 227.

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

150c ⟨*Instruction call* 149⟩+≡ (231) ◁150b 151▷

```
      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 65.

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

151 ⟨*Instruction call* 149⟩+≡ (231) ◁150c 152a▷

```
    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 227, SCRATCH2 227, get_next_code_byte 65, and push 63.

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

152a ⟨*Instruction call* 149⟩+≡ (231) ◁151 152b▷

```
.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
```

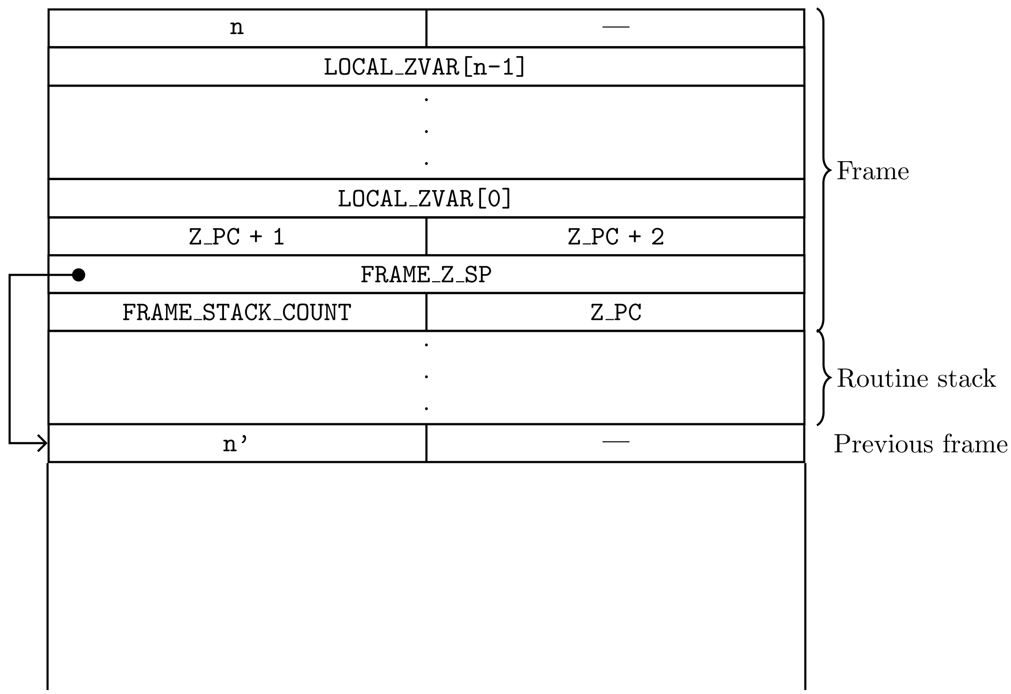
Uses LOCAL_ZVARS 227, OPERAND1 227, OPERAND_COUNT 227, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, and STOB 8b.

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.

152b ⟨*Instruction call* 149⟩+≡ (231) ◁152a

```
.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 227, FRAME_Z_SP 227, MOVB 8b, MOVW 8c, PULB 9b, SCRATCH2 227, STACK_COUNT 227, Z_SP 227, do_instruction 136, and push 63.



14.2 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.

153 ⟨*Instruction ret* 153⟩≡ (231) 154a ▷

```
instr_ret:
    SUBROUTINE

    MOVW    FRAME_Z_SP, Z_SP
    MOVB    FRAME_STACK_COUNT, STACK_COUNT
```

Defines:
`instr_ret`, used in chunks 133, 206a, and 207a.

Uses `FRAME_STACK_COUNT` 227, `FRAME_Z_SP` 227, `MOVB` 8b, `MOVW` 8c, `STACK_COUNT` 227, and `Z_SP` 227.

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.

154a ⟨*Instruction ret* 153⟩+≡ (231) ◁153 154b▷

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

Uses SCRATCH2 227 and pop 64.

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

154b ⟨*Instruction ret* 153⟩+≡ (231) ◁154a 154c▷

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

Uses ADDA 10b, GLOBAL_ZVARS_ADDR 227, MOVB 8b, SCRATCH1 227, SCRATCH2 227, SCRATCH3 227, and STOW 7.

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

154c ⟨*Instruction ret* 153⟩+≡ (231) ◁154b 154d▷

```
.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 227, SCRATCH2 227, SCRATCH3 227, SUBB 13a, and pop 64.

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

154d ⟨*Instruction ret* 153⟩+≡ (231) ◁154c 155▷

```
.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 227, FRAME_Z_SP 227, MOVB 8b, MOVW 8c, SCRATCH2 227, Z_PC 227, and pop 64.

Finally, we store the return value.

155 ⟨*Instruction ret* 153⟩+≡ (231) ◁154d

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

Uses `MOVW` 8c, `OPERAND0` 227, `SCRATCH2` 227, `STOB` 8b, `ZCODE_PAGE_VALID` 227, and `store_and_next` 182a.

Chapter 15

Objects

15.1 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).

15.2 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).

156 ⟨*Get object address* 156⟩≡ (231) 157a ▷

```
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 158–61, 163, 202b, 211, 213a, 218, and 219.
Uses SCRATCH2 227 and STOB 8b.

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.

157a ⟨*Get object address* 156⟩+≡ (231) ◁156 157b▷

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

Uses FIRST_OBJECT_OFFSET 229a and SCRATCH2 227.

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.

157b ⟨*Get object address* 156⟩+≡ (231) ◁157a

```
    .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 229a and SCRATCH2 227.

15.3 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.

158a ⟨*Remove object* 158a⟩≡ (231) 158b ▷

```
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 219 and 221a.

Uses `OBJECT_PARENT_OFFSET` 229a, `OPERAND0` 227, `SCRATCH2` 227, and `get_object_addr` 156.

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

158b ⟨*Remove object* 158a⟩+≡ (231) ◁ 158a 158c ▷

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

Uses `PSHW` 9a and `SCRATCH2` 227.

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

158c ⟨*Remove object* 158a⟩+≡ (231) ◁ 158b 159a ▷

```
    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` 229a, `SCRATCH2` 227, and `get_object_addr` 156.

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.

159a ⟨*Remove object* 158a⟩+≡ (231) ◁158c 159b▷
```
    CMP     OPERAND0              ; if (child_num != obj_num) loop
    BNE     .loop
```
Uses OPERAND0 227.

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

159b ⟨*Remove object* 158a⟩+≡ (231) ◁159a 159d▷
```
    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 229a, OBJECT_SIBLING_OFFSET 229a, PSHW 9a, PULW 9c, SCRATCH1 227, and SCRATCH2 227.

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

159c ⟨*Detach object* 159c⟩≡ (160a)
```
  .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 229a, PULW 9c, and SCRATCH2 227.

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.

159d ⟨*Remove object* 158a⟩+≡ (231) ◁159b 160a▷
```
  .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 229a, OPERAND0 227, SCRATCH2 227, and get_object_addr 156.

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.

160a ⟨*Remove object* 158a⟩+≡ (231) ◁159d

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

  ⟨Detach object 159c⟩
```

Uses PSHW 9a, PULW 9c, SCRATCH1 227, and SCRATCH2 227.

15.4 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.

160b ⟨*Print object in A* 160b⟩≡ (231)

```
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 93 and 209b.

Uses INCW 10a, MOVW 8c, OBJECT_PROPS_OFFSET 229a, SCRATCH1 227, SCRATCH2 227, get_object_addr 156, load_address 72b, and print_zstring 86.

15.5 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.

161 ⟨*Get attribute pointer and mask* 161⟩≡ (231) 162a ▷

```
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 204b, 210, and 221b.

Uses `INCW` 10a, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH1` 227, `SCRATCH2` 227, and `get_object_addr` 156.

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.

162a ⟨*Get attribute pointer and mask* 161⟩+≡ (231) ◁161 162b▷

```
    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 227, SCRATCH3 227, and STOW 7.

Finally, we load the attribute word into SCRATCH1.

162b ⟨*Get attribute pointer and mask* 161⟩+≡ (231) ◁162a

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

Uses SCRATCH1 227 and SCRATCH2 227.

15.6 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.

163 ⟨*Get property pointer* 163⟩≡ (231)

```
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 212, 213b, 216, and 220.

Uses `ADDW` 12a, `OBJECT_PROPS_OFFSET` 229a, `OPERAND0` 227, `SCRATCH1` 227, `SCRATCH2` 227, and `get_object_addr` 156.

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

164a ⟨*Get property number* 164a⟩≡ (231)

```
get_property_num:
    SUBROUTINE

    LDA      (SCRATCH2),Y
    AND      #$1F
    RTS
```

Defines:
`get_property_num`, used in chunks 212, 213b, 216, and 220.
Uses `SCRATCH2` 227.

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

164b ⟨*Get property length* 164b⟩≡ (231)

```
get_property_len:
    SUBROUTINE

    LDA      (SCRATCH2),Y
    ROR
    ROR
    ROR
    ROR
    ROR
    AND      #$07
    RTS
```

Defines:
`get_property_len`, used in chunks 165, 215, 217, and 220.
Uses `SCRATCH2` 227.

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

165 ⟨*Next property* 165⟩≡ (231)
```
  next_property:
      SUBROUTINE

      JSR     get_property_len
      TAX

  .loop:
      INY
      DEX
      BPL     .loop
      INY
      RTS
```
Defines:
  `next_property`, used in chunks 212, 213b, 216, and 220.
Uses `get_property_len` 164b.

Chapter 16

Saving and restoring the game

16.0.1 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:

166 ⟨*Insert save diskette* 166⟩≡ (230b) 167a ▷

```
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 172a and 175b.

Uses `SCRATCH2` 227, `STOW` 7, `dump_buffer_with_more` 79, `home` 75a, `print_ascii_string` 83b, and `sPleaseInsert` 167b.

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.

167a ⟨*Insert save diskette* 166⟩+≡ (230b) ◁166 169a▷

```
  .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` 81, `prompt_offset` 167b, `sPositionPrompt` 167b, `sSlotPrompt` 167b, and `save_position` 167b.

167b ⟨*Save diskette strings* 167b⟩≡ (230b)

```
  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 167–69.
- `sDrivePrompt`, used in chunk 169b.
- `sPleaseInsert`, used in chunk 166.
- `sPositionPrompt`, used in chunk 167a.
- `sReturnToBegin`, used in chunk 170a.
- `sSlotPrompt`, used in chunks 167–69.
- `save_drive`, used in chunks 169b and 250.
- `save_position`, used in chunks 167a and 170b.
- `save_slot`, used in chunks 168 and 169a.

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.

168 ⟨*Get prompted number from user* 168⟩≡ (230b)

```
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 11b, CH 226, CLREOL 226, INVFLG 226, RDKEY 226, SCRATCH2 227, STOW 7, cout_string 74, dump_buffer_line 78, dump_buffer_with_more 79, print_ascii_string 83b, prompt_offset 167b, sSlotPrompt 167b, and save_slot 167b.

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`.

169a ⟨*Insert save diskette* 166⟩+≡ (230b) ◁167a 169b▷

```
.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` 81, `iob` 129, `prompt_offset` 167b, `sSlotPrompt` 167b, and `save_slot` 167b.

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

169b ⟨*Insert save diskette* 166⟩+≡ (230b) ◁169a 170a▷

```
.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` 81, `iob` 129, `prompt_offset` 167b, `sDrivePrompt` 167b, `sSlotPrompt` 167b, and `save_drive` 167b.

Next, we prompt the user to start.

170a ⟨*Insert save diskette* 166⟩+≡ (230b) ◁169b 170b▷

```
.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 226, SCRATCH2 227, STOW 7, dump_buffer_line 78, dump_buffer_with_more 79, print_ascii_string 83b, and sReturnToBegin 167b.

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.

170b ⟨*Insert save diskette* 166⟩+≡ (230b) ◁170a

```
    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 11b, SCRATCH1 227, dump_buffer_with_more 79, and save_position 167b.

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

171 ⟨*Reinsert game diskette* 171⟩≡ (230b)

```
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 174c, 175a, and 178.
`sPressReturnToContinue`, never used.
`sReinsertGameDiskette`, never used.

Uses `RDKEY` 226, `SCRATCH2` 227, `STOB` 8b, `STOW` 7, `dump_buffer_line` 78, `dump_buffer_with_more` 79, `iob` 129, and `print_ascii_string` 83b.

16.0.2 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.

172a ⟨*Instruction save* 172a⟩≡ (230b) 172b ▷

```
instr_save:
    SUBROUTINE

    JSR     please_insert_save_diskette
```

Defines:
`instr_save`, used in chunk 133.
Uses `please_insert_save_diskette` 166.

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.

172b ⟨*Instruction save* 172a⟩+≡ (230b) ◁ 172a 172c ▷

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

Uses `BUFF_AREA` 227.

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

172c ⟨*Instruction save* 172a⟩+≡ (230b) ◁ 172b 173b ▷

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

Uses `SCRATCH2` 227, `STOW` 7, `Z_PC` 227, and `copy_data_to_buff` 173a.

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.

173a ⟨*Copy data to buff* 173a⟩≡ (230b)

```
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 172–74.
Uses BUFF_AREA 227 and SCRATCH2 227.

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.

173b ⟨*Instruction save* 172a⟩+≡ (230b) ◁172c 174a▷

```
    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 227, SCRATCH2 227, STACK_COUNT 227, STOW 7, copy_data_to_buff 173a, and write_next_sector 132.

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

174a ⟨*Instruction save* 172a⟩+≡ (230b) ◁173b 174b▷

```
    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 227, STOW 7, copy_data_to_buff 173a, and write_next_sector 132.

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.

174b ⟨*Instruction save* 172a⟩+≡ (230b) ◁174a 174c▷

```
    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 229a, MOVW 8c, SCRATCH2 227, SCRATCH3 227, and inc_sector_and_write 132.

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

174c ⟨*Instruction save* 172a⟩+≡ (230b) ◁174b 175a▷

```
    JSR     please_reinsert_game_diskette
    JMP     branch
```

Uses branch 184a and please_reinsert_game_diskette 171.

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

175a ⟨*Instruction save* 172a⟩+≡ (230b) ◁174c

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

Uses negated_branch 184a and please_reinsert_game_diskette 171.

16.0.3 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.

175b ⟨*Instruction restore* 175b⟩≡ (230b) 175c▷

```
instr_restore:
    SUBROUTINE

    JSR     please_insert_save_diskette
```

Defines:
instr_restore, used in chunk 133.
Uses please_insert_save_diskette 166.

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.

175c ⟨*Instruction restore* 175b⟩+≡ (230b) ◁175b 176a▷

```
    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 227 and read_next_sector 131.

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

176a ⟨*Instruction restore* 175b⟩+≡ (230b) ◁175c 176b▷

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

Uses `HEADER_FLAGS2` 229a.

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

176b ⟨*Instruction restore* 175b⟩+≡ (230b) ◁176a 177a▷

```
    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` 227, `SCRATCH2` 227, `STACK_COUNT` 227, `STOW` 7, `ZCODE_PAGE_VALID` 227, `Z_PC` 227, and `copy_data_from_buff` 176c.

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.

176c ⟨*Copy data from buff* 176c⟩≡ (230b)

```
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 176b and 177a.

Uses `BUFF_AREA` 227 and `SCRATCH2` 227.

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

177a ⟨*Instruction restore* 175b⟩+≡ (230b) ◁176b 177b▷

```
    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 227, STOW 7, copy_data_from_buff 176c, and read_next_sector 131.

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.

177b ⟨*Instruction restore* 175b⟩+≡ (230b) ◁177a 177c▷

```
    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 229a, MOVW 8c, SCRATCH2 227, SCRATCH3 227, and inc_sector_and_read 131.

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

177c ⟨*Instruction restore* 175b⟩+≡ (230b) ◁177b 178a▷

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

Uses HEADER_FLAGS2 229a.

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

178a ⟨*Instruction restore* 175b⟩+≡ (230b) ◁177c 178b▷

```
        JSR     please_reinsert_game_diskette
        JMP     branch
```

Uses `branch` 184a and `please_reinsert_game_diskette` 171.

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

178b ⟨*Instruction restore* 175b⟩+≡ (230b) ◁178a

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

Uses `negated_branch` 184a and `please_reinsert_game_diskette` 171.

Chapter 17

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

17.1 Instruction utilities

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

`illegal_opcode` hits a `BRK` instruction.

181 ⟨*Instruction illegal opcode* 181⟩≡ (231)

```
illegal_opcode:
    SUBROUTINE


    JSR     brk
```

Defines:
`illegal_opcode`, used in chunks 133, 137b, 139a, 141b, and 143.
Uses `brk` 59b.

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.

182a ⟨*Store and go to next instruction* 182a⟩≡ (230a)
```
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 212 and 217.
store_and_next, used in chunks 149, 155, 188, 189a, 193b, 195–99, and 213–16.
store_zero_and_next, used in chunks 212 and 216.
Uses SCRATCH2 227, STOB 8b, do_instruction 136, and store_var 147.

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

182b ⟨*Print zstring and go to next instruction* 182b⟩≡ (231)
```
print_zstring_and_next:
    SUBROUTINE

    JSR     print_zstring
    JMP     do_instruction
```
Defines:
print_zstring_and_next, used in chunks 208b and 209c.
Uses do_instruction 136 and print_zstring 86.

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

183a ⟨*Increment variable* 183a⟩≡ (231)
```
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 192c and 200a.
Uses INCW 10a, OPERAND0 227, PSHW 9a, PULW 9c, SCRATCH2 227, var_get 146, and var_put 148.

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

183b ⟨*Decrement variable* 183b⟩≡ (231)
```
dec_var:
    SUBROUTINE

    LDA     OPERAND0
    JSR     var_get
    SUBB    SCRATCH2, #$01
    JMP     inc_var_continue
```
Defines:
dec_var, used in chunks 193a and 199b.
Uses OPERAND0 227, SCRATCH2 227, SUBB 13a, and var_get 146.

17.1.1 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.

184a ⟨*Handle branch* 184a⟩≡ (230a) 184b ▷

```
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 174c, 178a, 200, 201, and 203b.
  `negated_branch`, used in chunks 175a, 178b, 200–204, and 211.
Uses `get_next_code_byte` 65.

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.

184b ⟨*Handle branch* 184a⟩+≡ (230a) ◁ 184a 185 ▷

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

.next:
    JMP      do_instruction
```

Uses `do_instruction` 136 and `get_next_code_byte` 65.

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.

185 ⟨*Handle branch* 184a⟩+≡ (230a) ◁184b 186a▷

```
.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` 227 and `get_next_code_byte` 65.

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.

186a ⟨*Handle branch* 184a⟩+≡ (230a) ◁185 186b▷

```
  .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 227, instr_rfalse 206b, and instr_rtrue 207a.

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.

186b ⟨*Handle branch* 184a⟩+≡ (230a) ◁186a 186c▷

```
  branch_to_offset:
      SUBROUTINE

      SUBB      SCRATCH2, #$01
```

Defines:
  branch_to_offset, used in chunk 205a.
Uses SCRATCH2 227 and SUBB 13a.

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

186c ⟨*Handle branch* 184a⟩+≡ (230a) ◁186b 187▷

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

Uses SCRATCH1 227 and SCRATCH2 227.

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.

187 ⟨*Handle branch* 184a⟩+≡ (230a) ◁186c

```
    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 227, SCRATCH2 227, ZCODE_PAGE_VALID 227, Z_PC 227, and do_instruction 136.

17.2 Data movement instructions

17.2.1 load

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

188a ⟨*Instruction load* 188a⟩≡ (231)

```
  instr_load:
      SUBROUTINE

      LDA       OPERAND0
      JSR       var_get
      JMP       store_and_next
```

Defines:
  `instr_load`, used in chunk 133.
Uses `OPERAND0` 227, `store_and_next` 182a, and `var_get` 146.

17.2.2 loadw

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

188b ⟨*Instruction loadw* 188b⟩≡ (231)

```
  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 133.
Uses `ADDW` 12a, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH2` 227, `get_next_code_word` 72a, `load_address` 72b, and `store_and_next` 182a.

17.2.3 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.

189a ⟨*Instruction loadb* 189a⟩≡ (231)

```
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 133.

Uses `ADDW` 12a, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH2` 227, `get_next_code_byte2` 70, `load_address` 72b, and `store_and_next` 182a.

17.2.4 store

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

189b ⟨*Instruction store* 189b⟩≡ (231)

```
instr_store:
    SUBROUTINE

    MOVW     OPERAND1, SCRATCH2
    LDA      OPERAND0

stretch_var_put:
    JSR      var_put
    JMP      do_instruction
```

Defines:
`instr_store`, used in chunk 133.
`stretch_var_put`, used in chunk 192a.

Uses `MOVW` 8c, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH2` 227, `do_instruction` 136, and `var_put` 148.

17.2.5 storew

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

190 ⟨*Instruction storew* 190⟩≡ (231)

```
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 133.

Uses `ADDW` 12a, `OPERAND0` 227, `OPERAND1` 227, `OPERAND2` 227, `SCRATCH2` 227, and `do_instruction` 136.

17.2.6 storeb

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

191a ⟨*Instruction storeb* 191a⟩≡ (231)

```
  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 133.
Uses ADDW 12a, OPERAND0 227, OPERAND1 227, OPERAND2 227, SCRATCH2 227, and do_instruction 136.

17.3 Stack instructions

17.3.1 pop

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

191b ⟨*Instruction pop* 191b⟩≡ (231)

```
  instr_pop:
      SUBROUTINE

      JSR       pop
      JMP       do_instruction
```

Defines:
instr_pop, used in chunk 133.
Uses do_instruction 136 and pop 64.

17.3.2 pull

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

192a ⟨*Instruction pull* 192a⟩≡ (231)

```
  instr_pull:
      SUBROUTINE

      JSR      pop
      LDA      OPERAND0
      JMP      stretch_var_put
```

Defines:
  instr_pull, used in chunk 133.
Uses OPERAND0 227, pop 64, and stretch_var_put 189b.

17.3.3 push

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

192b ⟨*Instruction push* 192b⟩≡ (231)

```
  instr_push:
      SUBROUTINE

      MOVW     OPERAND0, SCRATCH2
      JSR      push
      JMP      do_instruction
```

Defines:
  instr_push, used in chunk 133.
Uses MOVW 8c, OPERAND0 227, SCRATCH2 227, do_instruction 136, and push 63.

17.4 Decrements and increments

17.4.1 inc

inc increments the variable in the operand.

192c ⟨*Instruction inc* 192c⟩≡ (231)

```
  instr_inc:
      SUBROUTINE

      JSR      inc_var
      JMP      do_instruction
```

Defines:
  instr_inc, used in chunk 133.
Uses do_instruction 136 and inc_var 183a.

17.4.2 dec

dec decrements the variable in the operand.

193a ⟨*Instruction dec* 193a⟩≡ (231)
```
  instr_dec:
      SUBROUTINE

      JSR     dec_var
      JMP     do_instruction
```
Defines:
instr_dec, used in chunk 133.
Uses dec_var 183b and do_instruction 136.

17.5 Arithmetic instructions

17.5.1 add

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

193b ⟨*Instruction add* 193b⟩≡ (231)
```
  instr_add:
      SUBROUTINE

      ADDW    OPERAND0, OPERAND1, SCRATCH2
      JMP     store_and_next
```
Defines:
instr_add, used in chunk 133.
Uses ADDW 12a, OPERAND0 227, OPERAND1 227, SCRATCH2 227, and store_and_next 182a.

17.5.2 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.

194 ⟨*Instruction div* 194⟩≡ (231)

```
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 133.

Uses `MOVW` 8c, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH1` 227, `SCRATCH2` 227, `check_sign` 119b, and `divu16` 124.

17.5.3 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.

195 ⟨*Instruction mod* 195⟩≡ (231)

```
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 133.

Uses `MOVW` 8c, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH1` 227, `SCRATCH2` 227, `check_sign` 119b, `divu16` 124, and `store_and_next` 182a.

17.5.4 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.

196 ⟨*Instruction mul* 196⟩≡ (231)

```
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 133.

Uses `MOVW` 8c, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH1` 227, `SCRATCH2` 227, `check_sign` 119b, `mulu16` 121, `set_sign` 120, and `store_and_next` 182a.

17.5.5 random

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

197a ⟨*Instruction random* 197a⟩≡ (231)
```
  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 133.
Uses `INCW` 10a, `MOVW` 8c, `OPERAND0` 227, `SCRATCH1` 227, `SCRATCH2` 227, `divu16` 124, `get_random` 197b, and `store_and_next` 182a.

197b ⟨*Get random* 197b⟩≡ (231)
```
  get_random:
      SUBROUTINE

      ROL     RANDOM_VAL+1
      MOVW    RANDOM_VAL, SCRATCH2
      RTS
```
Defines:
  `get_random`, used in chunk 197a.
Uses `MOVW` 8c and `SCRATCH2` 227.

17.5.6 sub

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

197c ⟨*Instruction sub* 197c⟩≡ (231)
```
  instr_sub:
      SUBROUTINE

      SUBW    OPERAND0, OPERAND1, SCRATCH2
      JMP     store_and_next
```
Defines:
  `instr_sub`, used in chunk 133.
Uses `OPERAND0` 227, `OPERAND1` 227, `SCRATCH2` 227, `SUBW` 14a, and `store_and_next` 182a.

17.6 Logical instructions

17.6.1 and

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

198a ⟨*Instruction and* 198a⟩≡ (231)

```
  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 133.
Uses `OPERAND0` 227, `OPERAND1` 227, `SCRATCH2` 227, and `store_and_next` 182a.

17.6.2 not

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

198b ⟨*Instruction not* 198b⟩≡ (231)

```
  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 133.
Uses `OPERAND0` 227, `SCRATCH2` 227, and `store_and_next` 182a.

17.6.3 or

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

199a ⟨*Instruction or* 199a⟩≡ (231)

```
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 133.
Uses `OPERAND0` 227, `OPERAND1` 227, `SCRATCH2` 227, and `store_and_next` 182a.

17.7 Conditional branch instructions

17.7.1 dec_chk

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

199b ⟨*Instruction dec chk* 199b⟩≡ (231)

```
instr_dec_chk:
    SUBROUTINE

    JSR     dec_var
    MOVW    OPERAND1, SCRATCH1
    JMP     do_chk
```

Defines:
  `instr_dec_chk`, used in chunk 133.
Uses `MOVW` 8c, `OPERAND1` 227, `SCRATCH1` 227, `dec_var` 183b, and `do_chk` 200a.

17.7.2 inc_chk

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

200a ⟨*Instruction inc chk* 200a⟩≡ (231)
```
  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 199b.
instr_inc_chk, used in chunk 133.
stretch_to_branch, used in chunks 202–4.
Uses MOVW 8c, OPERAND1 227, SCRATCH1 227, SCRATCH2 227, branch 184a, cmp16 125b, inc_var 183a, and negated_branch 184a.

17.7.3 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.

200b ⟨*Instruction je* 200b⟩≡ (231) 201a ▷
```
  instr_je:
      SUBROUTINE

      LDX       OPERAND_COUNT
      DEX
      BNE       .check_second
      JSR       brk
```
Defines:
instr_je, used in chunk 133.
Uses OPERAND_COUNT 227 and brk 59b.

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

201a ⟨*Instruction je* 200b⟩+≡ (231) ◁200b 201b▷

```
  .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` 227, `OPERAND1` 227, and `branch` 184a.

Next we do the same with the third operand.

201b ⟨*Instruction je* 200b⟩+≡ (231) ◁201a 201c▷

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

  .check_next2:
      DEX
      BEQ       .neg_branch
```

Uses `OPERAND0` 227 and `branch` 184a.

And again with the fourth operand.

201c ⟨*Instruction je* 200b⟩+≡ (231) ◁201b

```
      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` 227, `branch` 184a, and `negated_branch` 184a.

17.7.4 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.

202a ⟨*Instruction jg* 202a⟩≡ (231)
```
instr_jg:
    SUBROUTINE

    MOVW      OPERAND0, SCRATCH1
    MOVW      OPERAND1, SCRATCH2
    JSR       cmp16
    BCC       stretch_to_branch
    JMP       negated_branch
```
Defines:
`instr_jg`, used in chunk 133.

Uses `MOVW` 8c, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH1` 227, `SCRATCH2` 227, `cmp16` 125b, `negated_branch` 184a, and `stretch_to_branch` 200a.

17.7.5 jin

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

202b ⟨*Instruction jin* 202b⟩≡ (231)
```
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 133.

Uses `OBJECT_PARENT_OFFSET` 229a, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH2` 227, `get_object_addr` 156, `negated_branch` 184a, and `stretch_to_branch` 200a.

17.7.6 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.

203a ⟨*Instruction jl* 203a⟩≡ (231)
```
instr_jl:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    MOVW     OPERAND1, SCRATCH1
    JSR      cmp16
    BCC      stretch_to_branch
    JMP      negated_branch
```
Defines:
  `instr_jl`, used in chunk 133.
Uses `MOVW` 8c, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH1` 227, `SCRATCH2` 227, `cmp16` 125b, `negated_branch` 184a, and `stretch_to_branch` 200a.

17.7.7 jz

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

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

203b ⟨*Instruction jz* 203b⟩≡ (231)
```
instr_jz:
    SUBROUTINE

    LDA      OPERAND0+1
    ORA      OPERAND0
    BEQ      take_branch
    JMP      negated_branch

take_branch:
    JMP      branch
```
Defines:
  `instr_jz`, used in chunk 133.
  `take_branch`, used in chunk 211.
Uses `OPERAND0` 227, `branch` 184a, and `negated_branch` 184a.

17.7.8 test

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

204a ⟨*Instruction test* 204a⟩≡ (231)

```
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 133.
Uses `MOVB` 8b, `OPERAND0` 227, `OPERAND1` 227, `SCRATCH1` 227, `SCRATCH2` 227, `cmpu16` 125a, `negated_branch` 184a, and `stretch_to_branch` 200a.

17.7.9 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.

204b ⟨*Instruction test attr* 204b⟩≡ (231)

```
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 133.
Uses `SCRATCH1` 227, `SCRATCH3` 227, `attr_ptr_and_mask` 161, `negated_branch` 184a, and `stretch_to_branch` 200a.

17.8 Jump and subroutine instructions

17.8.1 call

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

17.8.2 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.

205a ⟨*Instruction jump* 205a⟩≡ (231)

```
instr_jump:
    SUBROUTINE

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

Defines:
instr_jump, used in chunk 133.
Uses MOVW 8c, OPERAND0 227, SCRATCH2 227, SUBB 13a, and branch_to_offset 186b.

17.8.3 print_ret

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

205b ⟨*Instruction print ret* 205b⟩≡ (231)

```
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 133.
Uses buffer_char 81 and instr_rtrue 207a.

17.8.4 ret

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

17.8.5 ret_popped

`ret_popped` pops the stack and returns that value.

206a ⟨*Instruction ret popped* 206a⟩≡ (231)

```
  instr_ret_popped:
      SUBROUTINE

      JSR      pop
      MOVW     SCRATCH2, OPERAND0
      JMP      instr_ret
```

Defines:
`instr_ret_popped`, used in chunk 133.
Uses `MOVW` 8c, `OPERAND0` 227, `SCRATCH2` 227, `instr_ret` 153, and `pop` 64.

17.8.6 rfalse

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

206b ⟨*Instruction rfalse* 206b⟩≡ (231)

```
  instr_rfalse:
      SUBROUTINE

      LDA      #$00
      JMP      ret_a
```

Defines:
`instr_rfalse`, used in chunks 133 and 186a.
Uses `ret_a` 207a.

17.8.7 rtrue

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

207a ⟨*Instruction rtrue* 207a⟩≡ (231)

```
  instr_rtrue:
      SUBROUTINE

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

Defines:
`instr_rtrue`, used in chunks 133, 186a, and 205b.
`ret_a`, used in chunk 206b.
Uses `OPERAND0` 227 and `instr_ret` 153.

17.9 Print instructions

17.9.1 new_line

`new_line` prints `CRLF`.

207b ⟨*Instruction new line* 207b⟩≡ (231)

```
  instr_new_line:
      SUBROUTINE

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

Defines:
`instr_new_line`, used in chunk 133.
Uses `buffer_char` 81 and `do_instruction` 136.

17.9.2 print

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

208a ⟨*Instruction print* 208a⟩≡ (231)
```
instr_print:
    SUBROUTINE

    JSR       print_string_literal
    JMP       do_instruction
```
Defines:
instr_print, used in chunk 133.
Uses do_instruction 136.

17.9.3 print_addr

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

208b ⟨*Instruction print addr* 208b⟩≡ (231)
```
instr_print_addr:
    SUBROUTINE

    MOVW      OPERAND0, SCRATCH2
    JSR       load_address
    JMP       print_zstring_and_next
```
Defines:
instr_print_addr, used in chunk 133.
Uses MOVW 8c, OPERAND0 227, SCRATCH2 227, load_address 72b, and print_zstring_and_next 182b.

17.9.4 print_char

print_char prints the one-byte ASCII character in OPERAND0.

208c ⟨*Instruction print char* 208c⟩≡ (231)
```
instr_print_char:
    SUBROUTINE

    LDA       OPERAND0
    JSR       buffer_char
    JMP       do_instruction
```
Defines:
instr_print_char, used in chunk 133.
Uses OPERAND0 227, buffer_char 81, and do_instruction 136.

17.9.5 print_num

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

209a ⟨*Instruction print num* 209a⟩≡ (231)
```
  instr_print_num:
      SUBROUTINE

      MOVW      OPERAND0, SCRATCH2
      JSR       print_number
      JMP       do_instruction
```
Defines:
`instr_print_num`, used in chunk 133.
Uses `MOVW` 8c, `OPERAND0` 227, `SCRATCH2` 227, `do_instruction` 136, and `print_number` 127.

17.9.6 print_obj

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

209b ⟨*Instruction print obj* 209b⟩≡ (231)
```
  instr_print_obj:
      SUBROUTINE

      LDA       OPERAND0
      JSR       print_obj_in_A
      JMP       do_instruction
```
Defines:
`instr_print_obj`, used in chunk 133.
Uses `OPERAND0` 227, `do_instruction` 136, and `print_obj_in_A` 160b.

17.9.7 print_paddr

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

209c ⟨*Instruction print paddr* 209c⟩≡ (231)
```
  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 133.
Uses `MOVW` 8c, `OPERAND0` 227, `SCRATCH2` 227, `load_packed_address` 73,
and `print_zstring_and_next` 182b.

17.10 Object instructions

17.10.1 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.

210 ⟨*Instruction clear attr* 210⟩≡ (231)

```
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 133.

Uses `SCRATCH1` 227, `SCRATCH2` 227, `SCRATCH3` 227, `attr_ptr_and_mask` 161, and `do_instruction` 136.

17.10.2 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`).

211 ⟨*Instruction get child* 211⟩≡ (231)

```
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 218.

Uses `OBJECT_CHILD_OFFSET` 229a, `OPERAND0` 227, `SCRATCH2` 227, `get_object_addr` 156, `negated_branch` 184a, `store_var` 147, and `take_branch` 203b.

17.10.3 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.

212 ⟨*Instruction get next prop* 212⟩≡ (231)

```
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 133.

Uses `OPERAND1` 227, `get_property_num` 164a, `get_property_ptr` 163, `next_property` 165, `store_A_and_next` 182a, and `store_zero_and_next` 182a.

17.10.4 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.

213a ⟨*Instruction get parent* 213a⟩≡ (231)

```
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 133.
Uses `OBJECT_PARENT_OFFSET` 229a, `OPERAND0` 227, `SCRATCH2` 227, `get_object_addr` 156, and `store_and_next` 182a.

17.10.5 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.

213b ⟨*Instruction get prop* 213b⟩≡ (231) 214▷

```
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 133.

Uses `OPERAND1` 227, `get_property_num` 164a, `get_property_ptr` 163, and `next_property` 165.

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.

214 ⟨*Instruction get prop* 213b⟩+≡ (231) ◁213b 215▷

```
  .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 229a, OPERAND1 227, SCRATCH1 227, SCRATCH2 227, and store_and_next 182a.

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.

215 ⟨*Instruction get prop* 213b⟩+≡ (231) ◁214

```
.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 8c, SCRATCH1 227, SCRATCH2 227, brk 59b, get_property_len 164b, and store_and_next 182a.

17.10.6 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.

216 ⟨*Instruction get prop addr* 216⟩≡ (231)

```
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 133.

Uses `ADDAC` 11a, `INCW` 10a, `OPERAND1` 227, `SCRATCH2` 227, `SUBW` 14a, `get_property_num` 164a, `get_property_ptr` 163, `next_property` 165, `store_and_next` 182a, and `store_zero_and_next` 182a.

17.10.7 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.

217 ⟨*Instruction get prop len* 217⟩≡ (231)

```
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 133.

Uses `OPERAND0` 227, `SCRATCH2` 227, `get_property_len` 164b, and `store_A_and_next` 182a.

17.10.8 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`).

218 ⟨*Instruction get sibling* 218⟩≡ (231)

```
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 133.

Uses `OBJECT_SIBLING_OFFSET` 229a, `OPERAND0` 227, `get_object_addr` 156, and `push_and_check_obj` 211.

17.10.9 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.

219 ⟨*Instruction insert obj* 219⟩≡ (231)

```
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 133.

Uses OBJECT_CHILD_OFFSET 229a, OBJECT_PARENT_OFFSET 229a, OBJECT_SIBLING_OFFSET 229a, OPERAND0 227, OPERAND1 227, PSHW 9a, PULW 9c, SCRATCH2 227, do_instruction 136, get_object_addr 156, and remove_obj 158a.

17.10.10 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.

220 ⟨*Instruction put prop* 220⟩≡ (231)

```
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 133.

Uses OPERAND1 227, OPERAND2 227, SCRATCH2 227, brk 59b, do_instruction 136, get_property_len 164b, get_property_num 164a, get_property_ptr 163, and next_property 165.

17.10.11 remove_obj

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

221a ⟨*Instruction remove obj* 221a⟩≡ (231)

```
instr_remove_obj:
    SUBROUTINE

    JSR     remove_obj
    JMP     do_instruction
```

Defines:
`instr_remove_obj`, used in chunk 133.
Uses `do_instruction` 136 and `remove_obj` 158a.

17.10.12 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.

221b ⟨*Instruction set attr* 221b⟩≡ (231)

```
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 133.
Uses `SCRATCH1` 227, `SCRATCH2` 227, `SCRATCH3` 227, `attr_ptr_and_mask` 161, and `do_instruction` 136.

17.11 Other instructions

17.11.1 nop

`nop` does nothing.

222a ⟨*Instruction nop* 222a⟩≡ (231)
```
instr_nop:
    SUBROUTINE

    JMP       do_instruction
```
Defines:
`instr_nop`, used in chunk 133.
Uses `do_instruction` 136.

17.11.2 restart

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

222b ⟨*Instruction restart* 222b⟩≡ (231)
```
instr_restart:
    SUBROUTINE

    JSR       dump_buffer_with_more
    JMP       main
```
Defines:
`instr_restart`, used in chunk 133.
Uses `dump_buffer_with_more` 79 and `main` 53a.

17.11.3 restore

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

17.11.4 quit

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

223 ⟨*Instruction quit* 223⟩≡ (231)

```
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 133.
Uses `SCRATCH2` 227, `STOW` 7, `dump_buffer_with_more` 79, and `print_ascii_string` 83b.

17.11.5 save

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

17.11.6 sread

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

Chapter 18

The entire program

224a ⟨*boot1.asm* 224a⟩≡
```
        PROCESSOR 6502

    ⟨Macros 7⟩
    ⟨defines 225b⟩

        ORG         $0800

    ⟨BOOT1 16⟩
```

224b ⟨*boot2.asm* 224b⟩≡
```
        PROCESSOR 6502

    ⟨Macros 7⟩
    ⟨Apple ROM defines 226⟩
    ⟨RWTS defines 263⟩

    main    EQU     $0800

        ORG         $2300

    ⟨BOOT2 22⟩
```
Uses `main` 53a.

225a ⟨*main.asm* 225a⟩≡

```
        PROCESSOR 6502
```

⟨*Macros* 7⟩
⟨*defines* 225b⟩

```
        ORG       $0800
```

⟨*routines* 231⟩

225b ⟨*defines* 225b⟩≡ (224a 225a)
⟨*Apple ROM defines* 226⟩
⟨*Program defines* 227⟩
⟨*Table offsets* 229a⟩
⟨*variable numbers* 229b⟩

226 ⟨*Apple ROM defines* 226⟩≡ (224b 225b)

```
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 79, 93, and 168.
- `CLREOL`, used in chunks 79, 93, and 168.
- `COUT`, used in chunks 77 and 80b.
- `COUT1`, used in chunks 74, 76, and 80a.
- `CSW`, used in chunks 77 and 80b.
- `CV`, used in chunk 93.
- `GETLN1`, used in chunk 95.
- `HOME`, used in chunks 17 and 75a.
- `INIT`, used in chunk 23a.
- `INVFLG`, used in chunks 75b, 79, 93, and 168.
- `IWMDATAPTR`, never used.
- `IWMSECTOR`, never used.
- `IWMSLTNDX`, never used.
- `PROMPT`, used in chunk 75b.
- `RDKEY`, used in chunks 79, 168, 170a, and 171.
- `RDSECT_PTR`, never used.
- `SETKBD`, used in chunk 23a.
- `SETVID`, used in chunks 23a and 261c.
- `VTAB`, used in chunk 93.
- `WNDBTM`, used in chunks 75b and 79.
- `WNDLFT`, used in chunk 75b.
- `WNDTOP`, used in chunks 75, 79, and 95.
- `WNDWDTH`, used in chunks 75b and 80–83.

227 ⟨*Program defines 227*⟩≡ (225b)

```
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 76, 77, 81–83, 95, 131, 132, 172b, 173a, 175c, and 176c.
BUFF_END, used in chunks 76, 77, 80–83, and 95.
BUFF_LINE_LEN, used in chunks 82b and 83a.
CURR_DISK_BUFF_ADDR, never used.
CURR_LINE, used in chunks 75a, 79, and 95.
CURR_OPCODE, used in chunks 136, 139–41, and 143.
DEBUG_JUMP, used in chunks 23a, 135, and 263.
FIRST_Z_PAGE, used in chunks 55b and 69.
FRAME_STACK_COUNT, used in chunks 150a and 152–54.
FRAME_Z_SP, used in chunks 150a and 152–54.
GLOBAL_ZVARS_ADDR, used in chunks 60, 145, 147, and 154b.
HIGH_MEM_ADDR, used in chunks 61, 67, and 70.
LAST_Z_PAGE, used in chunks 55b, 61, 68, and 69.
LOCAL_ZVARS, used in chunks 145, 147, 151, 152a, 173b, and 176b.
LOCKED_ALPHABET, used in chunks 84, 86, 88, 89, 104, 105b, 107a, and 109a.
NEXT_PAGE_TABLE, used in chunks 54d, 55a, 61, and 69.
NUM_IMAGE_PAGES, used in chunks 57, 58a, 61, 66, and 70.
OPERAND0, used in chunks 95, 97, 99b, 100a, 102, 138d, 140a, 142, 149, 150b, 155, 158, 159, 161, 163, 183, 188–99, 201–9, 211, 213a, and 217–19.
OPERAND1, used in chunks 97–99, 101, 140b, 152a, 161, 188–91, 193–204, 212–14, 216, 219, and 220.
OPERAND2, used in chunks 190, 191a, and 220.
OPERAND3, never used.
OPERAND_COUNT, used in chunks 136, 138d, 141a, 142, 152a, and 200b.
PAGE_H_TABLE, used in chunks 54d, 55a, 67, 68, and 70.
PAGE_L_TABLE, used in chunks 54d, 55a, 67, 68, and 70.
PAGE_TABLE_INDEX, used in chunks 66, 67, and 70.
PAGE_TABLE_INDEX2, used in chunks 67 and 70.
PREV_PAGE_TABLE, used in chunks 54d, 55a, and 69.
PRINTER_CSW, used in chunks 54a, 77, and 80b.
RWTS, used in chunks 22, 130, 250, and 256.
SCRATCH1, used in chunks 56c, 58a, 67, 70, 101b, 104–107, 109, 110a, 112, 114–17, 119b, 121, 124, 125, 127, 130–32, 135, 145, 147, 152a, 154, 159–63, 170b, 186c, 187, 194–97, 199b, 200a, 202–4, 210, 214, 215, and 221b.
SCRATCH2, used in chunks 56c, 58a, 62–64, 66–70, 72–74, 79, 83b, 85, 89, 93, 103–106, 113–19, 121, 124, 125, 127, 130–32, 135, 137–45, 147–52, 154–64, 166, 168, 170–74, 176, 177, 182, 183, 185–200, 202–6, 208–11, 213–17, 219–21, and 223.
SCRATCH3, used in chunks 83b, 87a, 88, 90–92, 97–102, 104, 105, 107c, 109, 110, 112, 114–16, 121, 124, 127, 152a, 154, 162a, 174b, 177b, 204b, 210, and 221b.
SECTORS_PER_TRACK, used in chunks 23a, 130, and 263.
SHIFT_ALPHABET, used in chunks 84, 86, 88, and 89.
STACK_COUNT, used in chunks 54c, 63, 64, 152b, 153, 173b, and 176b.
TMP_Z_PC, used in chunk 136.
ZCHARS_H, used in chunks 85 and 89.
ZCHARS_L, used in chunks 85 and 89.
ZCHAR_SCRATCH1, used in chunks 54c, 99, 100, 105a, and 106b.
ZCHAR_SCRATCH2, used in chunks 104, 107–110, 112, 115a, and 116.
ZCODE_PAGE_ADDR, used in chunks 65, 66, and 92b.
ZCODE_PAGE_ADDR2, used in chunks 70 and 92b.

ZCODE_PAGE_VALID, used in chunks 54b, 65, 66, 70, 92b, 150a, 155, 176b, and 187.
ZCODE_PAGE_VALID2, used in chunks 54b, 67, 70, 73, 89, and 92b.
ZDECOMPRESS_STATE, used in chunks 85, 86, and 89.
Z_ABBREV_TABLE, used in chunks 60 and 89.
Z_PC, used in chunks 59c, 65–67, 92b, 136, 145, 150, 154d, 172c, 176b, and 187.
Z_PC2_H, used in chunks 70, 72b, 73, 89, and 92b.
Z_PC2_HH, used in chunks 70, 72b, 73, 89, and 92b.
Z_PC2_L, used in chunks 70, 72b, 73, 89, and 92b.
Z_SP, used in chunks 54c, 63, 64, 152b, and 153.

229a ⟨*Table offsets* 229a⟩≡ (225b)

```
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 157a.
HEADER_DICT_ADDR, used in chunk 113.
HEADER_FLAGS2, used in chunks 78, 80b, 95, 176a, and 177c.
HEADER_OBJECT_TABLE_ADDR, used in chunks 157b and 214.
HEADER_STATIC_MEM_BASE, used in chunks 174b and 177b.
OBJECT_CHILD_OFFSET, used in chunks 158c, 159b, 211, and 219.
OBJECT_PARENT_OFFSET, used in chunks 158a, 159c, 202b, 213a, and 219.
OBJECT_PROPS_OFFSET, used in chunks 160b and 163.
OBJECT_SIBLING_OFFSET, used in chunks 159, 218, and 219.

229b ⟨*variable numbers* 229b⟩≡ (225b)

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

Defines:
VAR_CURR_ROOM, used in chunk 93.
VAR_MAX_SCORE, used in chunk 93.
VAR_SCORE, used in chunk 93.

229c ⟨*Internal error string* 229c⟩≡ (231)
```
  sInternalError:
      DC       "ZORK INTERNAL ERROR!"
```
Defines:
`sInternalError`, never used.

230a ⟨*Instruction execution routines* 230a⟩≡ (231)

⟨*Instruction tables* 133⟩

⟨*Do instruction* 136⟩
⟨*Execute instruction* 135⟩
⟨*Handle 0op instructions* 137b⟩
⟨*Handle 1op instructions* 138a⟩
⟨*Handle 2op instructions* 140a⟩
⟨*Get const byte* 144a⟩
⟨*Get const word* 144b⟩
⟨*Get var content in A* 146⟩
⟨*Store to var A* 148⟩
⟨*Get var content* 145⟩
⟨*Store and go to next instruction* 182a⟩
⟨*Store var* 147⟩
⟨*Handle branch* 184a⟩

230b ⟨*Disk routines* 230b⟩≡ (231)

⟨*iob struct* 129⟩
⟨*Do RWTS on sector* 130⟩
⟨*Reading sectors* 131⟩
⟨*Writing sectors* 132⟩
⟨*Do reset window* 56b⟩
⟨*Print ASCII string* 83b⟩
⟨*Save diskette strings* 167b⟩
⟨*Insert save diskette* 166⟩
⟨*Get prompted number from user* 168⟩
⟨*Reinsert game diskette* 171⟩
⟨*Instruction save* 172a⟩
⟨*Copy data to buff* 173a⟩
⟨*Instruction restore* 175b⟩
⟨*Copy data from buff* 176c⟩

231 ⟨*routines* 231⟩≡ (225a)
⟨*main* 53a⟩

⟨*Instruction execution routines* 230a⟩
⟨*Instruction rtrue* 207a⟩
⟨*Instruction rfalse* 206b⟩
⟨*Instruction print* 208a⟩
⟨*Printing a string literal* 92b⟩
⟨*Instruction print ret* 205b⟩
⟨*Instruction nop* 222a⟩
⟨*Instruction ret popped* 206a⟩
⟨*Instruction pop* 191b⟩
⟨*Instruction new line* 207b⟩
⟨*Instruction jz* 203b⟩
⟨*Instruction get sibling* 218⟩
⟨*Instruction get child* 211⟩
⟨*Instruction get parent* 213a⟩
⟨*Instruction get prop len* 217⟩
⟨*Instruction inc* 192c⟩
⟨*Instruction dec* 193a⟩
⟨*Increment variable* 183a⟩
⟨*Decrement variable* 183b⟩
⟨*Instruction print addr* 208b⟩
⟨*Instruction illegal opcode* 181⟩
⟨*Instruction remove obj* 221a⟩
⟨*Remove object* 158a⟩
⟨*Instruction print obj* 209b⟩
⟨*Print object in A* 160b⟩
⟨*Instruction ret* 153⟩
⟨*Instruction jump* 205a⟩
⟨*Instruction print paddr* 209c⟩
⟨*Print zstring and go to next instruction* 182b⟩
⟨*Instruction load* 188a⟩
⟨*Instruction not* 198b⟩
⟨*Instruction jl* 203a⟩
⟨*Instruction jg* 202a⟩
⟨*Instruction dec chk* 199b⟩
⟨*Instruction inc chk* 200a⟩
⟨*Instruction jin* 202b⟩
⟨*Instruction test* 204a⟩
⟨*Instruction or* 199a⟩
⟨*Instruction and* 198a⟩
⟨*Instruction test attr* 204b⟩
⟨*Instruction set attr* 221b⟩
⟨*Instruction clear attr* 210⟩
⟨*Instruction store* 189b⟩
⟨*Instruction insert obj* 219⟩
⟨*Instruction loadw* 188b⟩
⟨*Instruction loadb* 189a⟩
⟨*Instruction get prop* 213b⟩

⟨*Instruction get prop addr* 216⟩
⟨*Instruction get next prop* 212⟩
⟨*Instruction add* 193b⟩
⟨*Instruction sub* 197c⟩
⟨*Instruction mul* 196⟩
⟨*Instruction div* 194⟩
⟨*Instruction mod* 195⟩
⟨*Instruction je* 200b⟩
⟨*Instruction call* 149⟩
⟨*Instruction storew* 190⟩
⟨*Instruction storeb* 191a⟩
⟨*Instruction put prop* 220⟩
⟨*Instruction sread* 97⟩
⟨*Skip separators* 102⟩
⟨*Separator checks* 103⟩
⟨*Get dictionary address* 113⟩
⟨*Match dictionary word* 114⟩
⟨*Instruction print char* 208c⟩
⟨*Instruction print num* 209a⟩
⟨*Print number* 127⟩
⟨*Print negative number* 128⟩
⟨*Instruction random* 197a⟩
⟨*Instruction push* 192b⟩
⟨*Instruction pull* 192a⟩
⟨*mulu16* 121⟩
⟨*divu16* 124⟩
⟨*Check sign* 119b⟩
⟨*Set sign* 120⟩
⟨*negate* 118⟩
⟨*Flip sign* 119a⟩
⟨*Get attribute pointer and mask* 161⟩
⟨*Get property pointer* 163⟩
⟨*Get property number* 164a⟩
⟨*Get property length* 164b⟩
⟨*Next property* 165⟩
⟨*Get object address* 156⟩
⟨*cmp16* 125b⟩
⟨*cmpu16* 125a⟩
⟨*Push* 63⟩
⟨*Pop* 64⟩
⟨*Get next code byte* 65⟩
⟨*Load address* 72b⟩
⟨*Load packed address* 73⟩
⟨*Get next code word* 72a⟩
⟨*Get next code byte 2* 70⟩
⟨*Set page first* 69⟩
⟨*Find index of page table* 68⟩
⟨*Print zstring* 86⟩
⟨*Printing a 10-bit ZSCII character* 92a⟩
⟨*Printing a space* 87b⟩

⟨*Printing a CRLF* 91c⟩
⟨*Shifting alphabets* 88⟩
⟨*Printing an abbreviation* 89⟩
⟨*A mod 3* 126⟩
⟨*A2 table* 91a⟩
⟨*Get alphabet* 84⟩
⟨*Get next zchar* 85⟩
⟨*ASCII to Zchar* 104⟩
⟨*Search nonalpha table* 111⟩
⟨*Get alphabet for char* 106a⟩
⟨*Z compress* 108⟩
⟨*Instruction restart* 222b⟩
⟨*Locate last RAM page* 62a⟩
⟨*Buffer a character* 81⟩
⟨*Dump buffer line* 78⟩
⟨*Dump buffer to printer* 77⟩
⟨*Dump buffer to screen* 76⟩
⟨*Dump buffer with more* 79⟩
⟨*Home* 75a⟩
⟨*Print status line* 93⟩
⟨*Output string to console* 74⟩
⟨*Read line* 95⟩
⟨*Reset window* 75b⟩
⟨*Disk routines* 230b⟩
⟨*Instruction quit* 223⟩
⟨*Internal error string* 229c⟩
⟨*brk* 59b⟩
⟨*Get random* 197b⟩

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

Chapter 19

Defined Chunks

Chapter 20

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.

235 ⟨*RWTS Prenibble routine* 235⟩≡ (264)

```
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 250.
Uses PRIMARY_BUFF 247a and SECONDARY_BUFF 247b.

237 ⟨*RWTS Write routine* 237⟩≡ (264)

```
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 250 and 259.

Uses `PRIMARY_BUFF` 247a, `SECONDARY_BUFF` 247b, `WRITE1` 239a, `WRITE2` 239a, `WRITE3` 239a, and `WRITE_XLAT_TABLE` 246b.

239a ⟨*RWTS Write bytes* 239a⟩≡ (264)

```
WRITE1:
    SUBROUTINE

    CLC

WRITE2:
    SUBROUTINE

    PHA
    PLA

WRITE3:
    SUBROUTINE

    STA     Q6H,X
    ORA     Q6L,X
    RTS
```

Defines:
WRITE1, used in chunk 237.
WRITE2, used in chunk 237.
WRITE3, used in chunk 237.

239b ⟨*RWTS Postnibble routine* 239b⟩≡ (264)

```
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 250.
Uses PRIMARY_BUFF 247a and SECONDARY_BUFF 247b.

240 ⟨*RWTS Read routine* 240⟩≡ (264)

```
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 250, 257, and 259.
read_error, used in chunk 242.
Uses ARM_MOVE_DELAY 245, PRIMARY_BUFF 247a, SECONDARY_BUFF 247b, and good_read 242.

242 ⟨*RWTS Read address* 242⟩≡ (264)

```
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 250, 257, and 259.
good_read, used in chunks 67, 70, and 240.
Uses read_error 240.

244 ⟨*RWTS Seek absolute* 244⟩≡ (264)

```
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 255a.
entry_off_end, never used.
Uses ARM_MOVE_DELAY 245, OFF_TABLE 246a, and ON_TABLE 246a.

245 ⟨*RWTS Arm move delay* 245⟩≡ (264)

```
  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 240, 244, and 250.

246a ⟨*RWTS Arm move delay tables* 246a⟩≡ (264)

```
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 244.
`ON_TABLE`, used in chunk 244.

246b ⟨*RWTS Write translate table* 246b⟩≡ (264)

```
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 237.

246c ⟨*RWTS Unused area* 246c⟩≡ (264)

```
    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
```

246d ⟨*RWTS Read translate table* 246d⟩≡ (264)

```
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.

247a ⟨*RWTS Primary buffer* 247a⟩≡ (264)

```
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 235, 237, 239b, 240, and 257.

247b ⟨*RWTS Secondary buffer* 247b⟩≡ (264)

```
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 235, 237, 239b, and 240.

248 ⟨*RWTS Write address header 248*⟩≡ (264)

```
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 259.
Uses WRITE_BYTE2 249a, WRITE_BYTE3 249a, and WRITE_DOUBLE_BYTE 249a.

249a ⟨*RWTS Write address header bytes* 249a⟩≡ (264)

```
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 248.
WRITE_BYTE3, used in chunk 248.
WRITE_DOUBLE_BYTE, used in chunk 248.

249b ⟨*RWTS Unused area 2* 249b⟩≡ (264)

```
    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
```

250 ⟨*RWTS Entry point 250*⟩≡ (264)

```
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 22.

Uses `ARM_MOVE_DELAY` 245, `PHYSECTOR` 261b, `POSTNIBBLE` 239b, `PRENIBBLE` 235, `READ` 240, `READ_ADDR` 242, `RWTS` 227, `WRITE` 237, `rwts_format` 257, `rwts_seek_track` 254, `rwts_set_track` 256, and `save_drive` 167b.

254 ⟨*RWTS seek track* 254⟩≡ (264)

```
    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 250 and 257.

Uses `rwts_move_arm` 255a.

255a ⟨*RWTS move arm* 255a⟩≡ (264)

```
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 254.
Uses `SEEKABS` 244 and `rwts_slot_x_to_y` 255b.

255b ⟨*RWTS Slot X to Y* 255b⟩≡ (264)

```
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 255a and 256.

256 ⟨*RWTS set track* 256⟩≡ (264)

```
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 250 and 257.

Uses `RWTS` 227 and `rwts_slot_x_to_y` 255b.

257 ⟨*RWTS Format disk* 257⟩≡ (264)

```
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 250.

Uses PRIMARY_BUFF 247a, READ 240, READ_ADDR 242, rwts_format_track 259, rwts_seek_track 254, and rwts_set_track 256.

259 ⟨*RWTS Format track* 259⟩≡ (264)

```
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 257.
Uses `READ` 240, `READ_ADDR` 242, `SECTOR_FLAGS` 261a, `WRITE` 237, and `WRITE_ADDR_HDR` 248.

261a ⟨*RWTS Sector flags* 261a⟩≡ (264)
```
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 259.

261b ⟨*RWTS Physical sector numbers* 261b⟩≡ (264)
```
PHYSECTOR:
        HEX     00 04 08 0C 01 05 09 0D
        HEX     02 06 0A 0E 03 07 0B 0F
```
Defines:
PHYSECTOR, used in chunk 250.

261c ⟨*RWTS Clobber language card* 261c⟩≡ (264)
```
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 226.

261d ⟨*RWTS Zero patch* 261d⟩≡ (264)
```
RWTS_ZERO_PATCH:
        SUBROUTINE

        STA     $1663
        STA     $1670
        STA     $1671
        RTS
```
Defines:
RWTS_ZERO_PATCH, never used.

261e ⟨*RWTS Patch 2* 261e⟩≡ (264)
```
RWTS_PATCH_2:
        SUBROUTINE

        JSR     $135B
        STY     $16B7
        RTS
```
Defines:
RWTS_PATCH_2, never used.

262 ⟨*RWTS Disk full error patch* 262⟩≡ (264)

```
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.

263 ⟨*RWTS defines* 263⟩≡ (224b)

```
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 227 and SECTORS_PER_TRACK 227.

264 ⟨*RWTS routines* 264⟩≡ (25b)
⟨*RWTS Prenibble routine* 235⟩
⟨*RWTS Write routine* 237⟩
⟨*RWTS Write bytes* 239a⟩
⟨*RWTS Postnibble routine* 239b⟩
⟨*RWTS Read routine* 240⟩
⟨*RWTS Read address* 242⟩
⟨*RWTS Seek absolute* 244⟩
⟨*RWTS Arm move delay* 245⟩
⟨*RWTS Arm move delay tables* 246a⟩
⟨*RWTS Write translate table* 246b⟩
⟨*RWTS Unused area* 246c⟩
⟨*RWTS Read translate table* 246d⟩
⟨*RWTS Primary buffer* 247a⟩
⟨*RWTS Secondary buffer* 247b⟩
⟨*RWTS Write address header* 248⟩
⟨*RWTS Write address header bytes* 249a⟩
⟨*RWTS Unused area 2* 249b⟩
⟨*RWTS Entry point* 250⟩
⟨*RWTS seek track* 254⟩
⟨*RWTS move arm* 255a⟩
⟨*RWTS Slot X to Y* 255b⟩
⟨*RWTS set track* 256⟩
⟨*RWTS Format disk* 257⟩
⟨*RWTS Format track* 259⟩
⟨*RWTS Sector flags* 261a⟩
⟨*RWTS Physical sector numbers* 261b⟩
⟨*RWTS Clobber language card* 261c⟩
⟨*RWTS Zero patch* 261d⟩
⟨*RWTS Patch 2* 261e⟩
⟨*RWTS Disk full error patch* 262⟩

Chapter 21

Index