# The Zork I Z-machine Interpreter

Robert Baruch

# Contents

# Chapter 1

# Zork I

## 1.1 Introduction

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

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

- Zork I, revision 15 (`ZorkI_r15_4amCrack`)

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

## 1.2 About this document

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

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

The assembly code is assembled using `dasm`.

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

# Chapter 2

# Programming techniques

## 2.1 Zero page temporaries

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

## 2.2 Tail calls

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

## 2.3 Unconditional branches

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

## 2.4 Stretchy branches

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

## 2.5 Shared code

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

## 2.6 Macros

The original Infocom source code uses macros for moving data around, and we will adopt these macros (with different names) to make our assembly language listings a little less verbose. We'll also make them all 4 characters long to differentiate them from 6502 mnemonics.

STOW stores a 16-bit literal value to a memory location in little-endian order. This is the same as MOVEI in the original Infocom source code.

5 ⟨*Macros* 5⟩≡ 6a ▷

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

Defines:
STOW, used in chunks 20–22, 42, 75b, 79, 82, 85, 103, 108b, 117a, and 165.

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

6a ⟨*Macros* 5⟩+≡ ◁5 6b▷

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

Defines:
MOVB, used in chunks 68a, 107, 108, and 146a.
STOB, used in chunks 20, 21c, 85, 104a, 106a, and 109.

MOVW moves a 16-bit value from one memory location to the another. This is the same as MOVEW in the original Infocom source code.

6b ⟨*Macros* 5⟩+≡ ◁6a 6c▷

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

Defines:
MOVW, used in chunks 22b, 79, 82, 104a, 106–109, 115, 131b, 134b, 136–39, 141b, 142a, 144a, 145a, 147a, 148a, 150, 151, and 157.

PSHW is a macro that pushes a 16-bit value in memory to the stack. This is the same as PUSHW in the original Infocom source code.

6c ⟨*Macros* 5⟩+≡ ◁6b 7a▷

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

Defines:
PSHW, used in chunks 79, 82, 125a, and 161.

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

7a ⟨*Macros* 5⟩+≡ ◁6c 7b▷

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

Defines:
PULB, used in chunk 106b.

PULW is a macro that pulls a 16-bit value from the stack to memory. This is the same as PULLW in the original Infocom source code.

7b ⟨*Macros* 5⟩+≡ ◁7a 7c▷

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

Defines:
PULW, used in chunks 79, 82, 125a, and 161.

INCW is a macro that increments a 16-bit value in memory. This is the same as INCW in the original Infocom source code.

7c ⟨*Macros* 5⟩+≡ ◁7b 7d▷

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

Defines:
INCW, used in chunks 115, 116, 125a, 139a, and 158.

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

7d ⟨*Macros* 5⟩+≡ ◁7c 8a▷

```
    MACRO ADDA
        CLC
        ADDAC   {1}
    ENDM
```

Defines:
ADDA, used in chunks 73 and 108b.
Uses ADDAC 8a.

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

8a ⟨*Macros* 5⟩+≡ ◁7d 8b▷

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

Defines:
ADDAC, used in chunks 7d and 158.

ADDB is a macro that adds an 8-bit immediate value, or the 8-bit contents of memory, to a 16-bit memory location. This is the same as ADDB in the original Infocom source code. The immediate value is the second argument.

8b ⟨*Macros* 5⟩+≡ ◁8a 8c▷

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

Defines:
ADDB, never used.

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

8c ⟨*Macros* 5⟩+≡ ◁8b 9a▷

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

Defines:
ADDB2, used in chunks 74 and 75a.

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

9a ⟨*Macros* 5⟩+≡ ◁8c 9b▷

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

Defines:
ADDW, used in chunks 58, 72, 118, 130–33, and 135b.
Uses ADDWC 9b.

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.

9b ⟨*Macros* 5⟩+≡ ◁9a 9c▷

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

Defines:
ADDWC, used in chunks 9a and 79.

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.

9c ⟨*Macros* 5⟩+≡ ◁9b 10a▷

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

Defines:
SUBB, used in chunks 75a, 108c, 125b, 128b, and 147a.

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

10a ⟨*Macros* 5⟩+≡ ◁9c 10b▷

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

Defines:
SUBB2, used in chunk 74b.

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.

10b ⟨*Macros* 5⟩+≡ ◁10a 10c▷

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

Defines:
SUBW, used in chunks 76, 77a, 139b, and 158.

ROLW rotates a 16-bit memory location left.

10c ⟨*Macros* 5⟩+≡ ◁10b 11▷

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

Defines:
ROLW, used in chunk 82.

RORW rotates a 16-bit memory location right.

11 ⟨*Macros* 5⟩+≡ ◁10c

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

Defines:
RORW, used in chunk 79.

# Chapter 3

# The boot process

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

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

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

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

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

## 3.1 BOOT1

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

13a ⟨*BOOT1* 13a⟩≡ 15d ▷

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

  ⟨Read BOOT2 from disk 13c⟩
  ⟨Jump to BOOT2 18⟩
  ⟨BOOT1 parameters 13b⟩
```

Defines:
  BOOT1, used in chunk 17.

13b ⟨*BOOT1 parameters* 13b⟩≡ (13a)

```
      ORG     $08FD

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

Defines:
  BOOT1_SECTOR_NUM, used in chunks 14, 15, and 17.
  BOOT1_WRITE_ADDR, used in chunks 14–17.

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

13c ⟨*Read BOOT2 from disk* 13c⟩≡ (13a) 16 ▷

```
  ⟨Skip initialization if BOOT1 already initialized 14a⟩
  .init_vars:
  ⟨Initialize BOOT1 14b⟩

  .already_initted:
  ⟨Set up parameters for reading a sector 15a⟩
      JMP     (RDSECT_PTR)
```

Uses RDSECT_PTR 167.

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

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

14a ⟨*Skip initialization if BOOT1 already initialized* 14a⟩≡ (13c)

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

Uses `IWMDATAPTR` 167.

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

14b ⟨*Initialize BOOT1* 14b⟩≡ (13c) 14c ▷

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

Uses `IWMSLTNDX` 167 and `RDSECT_PTR` 167.

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

14c ⟨*Initialize BOOT1* 14b⟩+≡ (13c) ◁ 14b

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

Uses `BOOT1_SECTOR_NUM` 13b and `BOOT1_WRITE_ADDR` 13b.

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

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

15a ⟨*Set up parameters for reading a sector* 15a⟩≡ (13c) 15c▷
```
      LDX     BOOT1_SECTOR_NUM
      BMI     .go_to_boot2      ; Are we done?
```
Uses BOOT1_SECTOR_NUM 13b.

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

15b ⟨*BOOT1 sector translation table* 15b⟩≡ (15d)
```
      ORG     $084D

  BOOT1_SECTOR_TRANSLATE_TABLE:
      HEX     00 0D 0B 09 07 05 03 01
      HEX     0E 0C 0A 08 06 04 02 0F
```
Defines:
BOOT1_SECTOR_TRANSLATE_TABLE, used in chunks 15c and 17.

15c ⟨*Set up parameters for reading a sector* 15a⟩+≡ (13c) ◁15a 15e▷
```
      LDA     BOOT1_SECTOR_TRANSLATE_TABLE,X
      STA     IWMSECTOR
```
Uses BOOT1_SECTOR_TRANSLATE_TABLE 15b and IWMSECTOR 167.

15d ⟨*BOOT1* 13a⟩+≡ ◁13a 17▷
⟨*BOOT1 sector translation table* 15b⟩

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

15e ⟨*Set up parameters for reading a sector* 15a⟩+≡ (13c) ◁15c
```
      DEC     BOOT1_SECTOR_NUM
      LDA     BOOT1_WRITE_ADDR+1
      STA     IWMDATAPTR+1
      DEC     BOOT1_WRITE_ADDR+1
      LDX     IWMSLTNDX
```
Uses BOOT1_SECTOR_NUM 13b, BOOT1_WRITE_ADDR 13b, IWMDATAPTR 167, and IWMSLTNDX 167.

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

16 ⟨*Read BOOT2 from disk* 13c⟩+≡ (13a) ◁13c

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

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

    JSR     SETKBD
    JSR     SETVID
    JSR     INIT

    ; Go to BOOT2!

    LDX     IWMSLTNDX
    JMP     (BOOT1_WRITE_ADDR)
```

Uses BOOT1_WRITE_ADDR 13b, INIT 167, IWMSLTNDX 167, SETKBD 167, and SETVID 167.

17 ⟨*BOOT1* 13a⟩+≡ ◁15d

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

    LDA     IWMDATAPTR+1
    CMP     #$09
    BNE     .already_initted

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

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

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

    CLC
    LDA     BOOT1_WRITE_ADDR+1
    ADC     BOOT1_SECTOR_NUM
    STA     BOOT1_WRITE_ADDR+1

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

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

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

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

```
    JMP     (RDSECT_PTR)

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

    INC     BOOT1_WRITE_ADDR+1
    INC     BOOT1_WRITE_ADDR+1

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

    JSR     SETKBD
    JSR     SETVID
    JSR     INIT

    ; Go to BOOT2!

    LDX     IWMSLTNDX
    JMP     (BOOT1_WRITE_ADDR)

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

Uses BOOT1 13a, BOOT1_SECTOR_NUM 13b, BOOT1_SECTOR_TRANSLATE_TABLE 15b, BOOT1_WRITE_ADDR 13b, INIT 167, IWMDATAPTR 167, IWMSECTOR 167, IWMSLTNDX 167, RDSECT_PTR 167, SETKBD 167, and SETVID 167.

18 ⟨*Jump to BOOT2* 18⟩≡ (13a)

## 3.2 BOOT2

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

# Chapter 4

# The main program

This is the Z-machine proper.

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

19a ⟨*main* 19a⟩≡ (171) 19b ▷

```
main:
    SUBROUTINE

    CLD
    LDA      #$00
    LDX      #$80

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

Defines:
`main`, used in chunks 22b, 23, 30, 32, and 164b.

And we reset the 6502 stack pointer.

19b ⟨*main* 19a⟩+≡ (171) ◁ 19a 20 ▷

```
    LDX      #$FF
    TXS
```

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

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

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

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

There are two other page tables, NEXT_PAGE_TABLE and PREV_PAGE_TABLE, which are set to $2300 and $2380, respectively. These are used to link Z-machine memory pages together.

20 ⟨*main* 19a⟩+≡ (171) ◁19b 21a▷

```
.set_vars:
    ; Historical note: Setting PRINTER_CSW was originally a call to SINIT,
    ; "system-dependent initialization".
    LDA      #$C1
    STA      PRINTER_CSW+1
    LDA      #$00
    STA      PRINTER_CSW
    LDA      #$00
    STA      ZCODE_PAGE_VALID
    STA      ZCODE_PAGE_VALID2
    STOB     #$01, STACK_COUNT
    STOW     #$03E8, Z_SP
    STOB     #$FF, DAT_00d9   ; ZSTBUI+6
    STOW     #$2200, PAGE_L_TABLE
    STOW     #$2280, PAGE_H_TABLE
    STOW     #$2300, NEXT_PAGE_TABLE
    STOW     #$2380, PREV_PAGE_TABLE
```

Uses NEXT_PAGE_TABLE 168, PAGE_H_TABLE 168, PAGE_L_TABLE 168, PREV_PAGE_TABLE 168, PRINTER_CSW 168, STACK_COUNT 168, STOB 6a, STOW 5, ZCODE_PAGE_VALID 168, ZCODE_PAGE_VALID2 168, and Z_SP 168.

Next, we initialize the page tables. This zeros out PAGE_L_TABLE and PAGE_H_TABLE, and then sets up the next and previous page tables. NEXT_PAGE_TABLE is initialized tp 01 02 03 ... 7F 80 and so on, while PREV_PAGE_TABLE is initialized to FF 00 01 ... 7D 7E. FF, in fact, is a marker for invalid page.

21a ⟨*main* 19a⟩+≡ (171) ◁20 21b▷

```
    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
```

Uses NEXT_PAGE_TABLE 168, PAGE_H_TABLE 168, PAGE_L_TABLE 168, and PREV_PAGE_TABLE 168.

21b ⟨*main* 19a⟩+≡ (171) ◁21a 21c▷

```
    DEY
    LDA     #$FF
    STA     (NEXT_PAGE_TABLE),Y
```

Uses NEXT_PAGE_TABLE 168.

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

21c ⟨*main* 19a⟩+≡ (171) ◁21b 21d▷

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

Uses LAST_Z_PAGE 168, STOB 6a, and STOW 5.

Then we clear the screen.

21d ⟨*main* 19a⟩+≡ (171) ◁21c 22b▷

```
    JSR     do_reset_window
```

Uses do_reset_window 22a.

22a ⟨*Do reset window* 22a⟩≡

```
do_reset_window:
    JSR     reset_window
    RTS
```

Defines:
`do_reset_window`, used in chunk 21d.
Uses `reset_window` 38.

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

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

22b ⟨*main* 19a⟩+≡ (171) ◁21d 23▷

```
.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` 6b, `SCRATCH1` 168, `SCRATCH2` 168, `STOW` 5, and `main` 19a.

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

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

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

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

23 ⟨*main* 19a⟩+≡ (171) ◁22b 24a▷

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

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

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

    BCC     .no_error2
    JMP     main
```

```
  .no_error2:
      PLA
      JMP      .read_another_sector
```
Uses NUM_IMAGE_PAGES 168, SCRATCH1 168, SCRATCH2 168, and main 19a.

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.

24a ⟨*main* 19a⟩+≡ (171) ◁23 24d▷
```
  .check_bit_0_flag:
      LDY      #$01
      LDA      (Z_HEADER_ADDR),Y
      AND      #$01
      EOR      #$01
      BEQ      .brk
```
Uses brk 24c.

24b ⟨*die* 24b⟩≡
```
  .brk:
      JSR      brk
```
Uses brk 24c.

24c ⟨*brk* 24c⟩≡
```
  brk:
      BRK
```
Defines:
brk, used in chunks 24, 26a, 123, 142b, 157, and 162.

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

24d ⟨*main* 19a⟩+≡ (171) ◁24a 25▷
```
  .store_initial_z_pc:
      LDY      #$07
      LDA      (Z_HEADER_ADDR),Y
      STA      Z_PC
      DEY
      LDA      (Z_HEADER_ADDR),Y
      STA      Z_PC+1
      LDA      #$00
      STA      Z_PC+2
```
Uses Z_PC 168.

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

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

25 ⟨*main* 19a⟩+≡ (171) ◁24d 26a▷

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

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

Uses GLOBAL_ZVARS_ADDR 168 and Z_ABBREV_TABLE 168.

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

For Zork I, AFTER_Z_IMAGE_ADDR is $7400.

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

26a ⟨*main* 19a⟩+≡ (171) ◁25 26b▷

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

Uses AFTER_Z_IMAGE_ADDR 168, NUM_IMAGE_PAGES 168, and brk 24c.

We then store the difference as the last Z-image page in 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.

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

26b ⟨*main* 19a⟩+≡ (171) ◁26a

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

Uses LAST_Z_PAGE 168, NEXT_PAGE_TABLE 168, NUM_PAGE_TABLE_ENTRIES 168, and do_instruction 90.

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`.

27 ⟨*Locate last RAM page* 27⟩≡

```
locate_last_ram_page:
    SUBROUTINE

    MOV     $#C0, SCRATCH2+1
    MOV     #$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 `SCRATCH2` 168.

# Chapter 5

# Z-code

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

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

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

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

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

28 ⟨*Get next code byte* 28⟩≡ 29▷

```
get_next_code_byte:
    LDA     ZCODE_PAGE_VALID
    BEQ     .zcode_page_invalid
    LDY     Z_PC
    LDA     (ZCODE_PAGE_ADDR),Y
    INY
    STY     Z_PC
    BEQ     .invalidate_zcode_page
    RTS

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

```
    INC     Z_PC+1
    BNE     .end
    INC     Z_PC+2

.end:
    RTS
```

Defines:
get_next_code_byte, used in chunks 31, 90, 91a, 98, 99, 101, 104c, 105, 126, and 127.
Uses ZCODE_PAGE_ADDR 168, ZCODE_PAGE_VALID 168, and Z_PC 168.

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

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

29 ⟨*Get next code byte* 28⟩+≡ ◁28

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

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

.adjust_page_link_tables:
    JSR     adjust_page_link_tables
    CLC
    LDA     PAGE_TABLE_INDEX
    ADC     NUM_IMAGE_PAGES

⟨Set page address 31⟩
```

Uses NUM_IMAGE_PAGES 168, PAGE_TABLE_INDEX 168, SCRATCH2 168, and Z_PC 168.

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

30 ⟨*Not found in page table* 30⟩≡

```
.not_found_in_page_table:
    CMP     DAT_0097
    BNE     .read_from_disk
    LDA     #$00
    STA     ZCODE_PAGE_VALID2

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

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

Uses AFTER_Z_IMAGE_ADDR 168, PAGE_H_TABLE 168, PAGE_L_TABLE 168, PAGE_TABLE_INDEX 168, SCRATCH1 168, SCRATCH2 168, ZCODE_PAGE_VALID2 168, Z_PC 168, and main 19a.

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

31 ⟨*Set page address* 31⟩≡ (29)

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

Uses ZCODE_PAGE_ADDR 168, ZCODE_PAGE_VALID 168, and get_next_code_byte 28.

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

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

32 ⟨*Get next code byte 2* 32⟩≡

```
get_next_code_byte2:
    SUBROUTINE

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

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

.end:
    RTS

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

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

.adjust_page_link_tables:
    JSR      adjust_page_link_tables
    CLC
```

```
    LDA     PAGE_TABLE_INDEX2
    ADC     NUM_IMAGE_PAGES

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

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

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

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

Defines:
get_next_code_byte2, used in chunks 34a and 131a.

Uses AFTER_Z_IMAGE_ADDR 168, NUM_IMAGE_PAGES 168, PAGE_H_TABLE 168, PAGE_L_TABLE 168, PAGE_TABLE_INDEX 168, PAGE_TABLE_INDEX2 168, SCRATCH1 168, SCRATCH2 168, ZCODE_PAGE_ADDR2 168, ZCODE_PAGE_VALID 168, ZCODE_PAGE_VALID2 168, Z_PC2_H 168, Z_PC2_HH 168, Z_PC2_L 168, and main 19a.

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`.

34a ⟨*Get next code word* 34a⟩≡

```
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 48 and 130b.
Uses `SCRATCH2` 168 and `get_next_code_byte2` 32.

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

34b ⟨*Load address* 34b⟩≡

```
load_address:
    SUBROUTINE

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

Defines:
`load_address`, used in chunks 115, 130b, 131a, and 150b.
Uses `SCRATCH2` 168, `Z_PC2_H` 168, `Z_PC2_HH` 168, and `Z_PC2_L` 168.

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

35 ⟨*Load packed address* 35⟩≡

```
invalidate_zcode_page2:
    SUBROUTINE

    LDA     #$00
    STA     ZCODE_PAGE_VALID2
    RTS

load_packed_address:
    SUBROUTINE

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

Defines:
invalidate_zcode_page2, never used.
load_packed_address, used in chunks 52 and 151c.
Uses SCRATCH2 168, ZCODE_PAGE_VALID2 168, Z_PC2_H 168, Z_PC2_HH 168, and Z_PC2_L 168.

# Chapter 6

# I/O

## 6.1 Strings and output

### 6.1.1 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.

36 ⟨*Output string to console* 36⟩≡

```
cout_string:
    SUBROUTINE

    LDY      #$00

.loop:
```

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

Defines:
`cout_string`, used in chunk 42.
Uses `COUT1` 167 and `SCRATCH2` 168.

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.

37 ⟨*Home* 37⟩≡

```
  home:
      SUBROUTINE

      JSR       HOME
      LDA       WNDTOP
      STA       CURR_LINE
      RTS
```

Defines:
`home`, used in chunk 38.
Uses `CURR_LINE` 168, `HOME` 167, and `WNDTOP` 167.

The `reset_window` routine tests 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.

38 ⟨*Reset window* 38⟩≡

```
reset_window:
    SUBROUTINE

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

Defines:
`reset_window`, used in chunk 22a.

Uses `INVFLG` 167, `PROMPT` 167, `WNDBTM` 167, `WNDLFT` 167, `WNDTOP` 167, `WNDWDTH` 167, and `home` 37.

When printing to the screen, Zork breaks lines between words. To do this, we buffer characters into the KBD_INPUT_AREA, which starting 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.

39 ⟨*Dump buffer to screen* 39⟩≡

```
dump_buffer_to_screen:
    LDX     #$00

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

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

Defines:
  dump_buffer_to_screen, used in chunk 41.
Uses BUFF_END 168, COUT1 167, and KBD_INPUT_AREA 168.

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.

40 ⟨*Dump buffer to printer* 40⟩≡

```
printer_card_initialized_flag:
    BYTE     00

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

.printer_set_80_column_output:
    LDA      #$09      ; ctrl-I
    JSR      COUT
    LDA      #$91      ; 'Q'
    STA      $0779     ; Scratchpad RAM for slot 1.
    LDA      #$B8      ; '8'
```

```
    JSR     COUT
    LDA     #$B0      ; '0'
    JSR     COUT
    LDA     #$CE      l 'N'
    JSR     COUT

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

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

Defines:
  `dump_buffer_to_printer`, used in chunks 41 and 56.
  `printer_card_initialized_flag`, never used.
Uses `BUFF_END` 168, `COUT` 167, `CSW` 167, `KBD_INPUT_AREA` 168, and `PRINTER_CSW` 168.

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`.

41 ⟨*Dump buffer line* 41⟩≡

```
dump_buffer_line:
    LDY     #$11
    LDA     (Z_HEADER),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 43a and 56.
Uses `dump_buffer_to_printer` 40 and `dump_buffer_to_screen` 39.

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.

42 ⟨*Dump buffer with more* 42⟩≡ 43a ▷

```
string_more:
    DC       "[MORE]"

dump_buffer_with_more:
    SUBROUTINE

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

    STOW     string_more, SCRATCH2
    LDX      #6

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

    LDA      #$FF
    STA      INVFLG

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

.good_to_go:
```

Defines:
dump_buffer_with_more, used in chunks 45, 46b, 164b, and 165.

Uses CH 167, CLREOL 167, CURR_LINE 168, INVFLG 167, RDKEY 167, SCRATCH2 168, STOW 5, WNDBTM 167, WNDTOP 167, and cout_string 36.

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.

43a ⟨*Dump buffer with more* 42⟩+≡ ◁42 43b▷

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

    .skip_newline:
```

Uses `BUFF_END` 168, `COUT1` 167, `WNDWDTH` 167, and `dump_buffer_line` 41.

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.

43b ⟨*Dump buffer with more* 42⟩+≡ ◁43a 44▷

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

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

        LDA     #$8D
        JSR     COUT

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

    .reset_buffer_end:
```

Uses `COUT` 167, `CSW` 167, and `PRINTER_CSW` 168.

The last step is to set BUFF_END to zero.

44 ⟨*Dump buffer with more 42*⟩+≡ ◁43b

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

Uses buffer_char_set_buffer_end 45.

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.

45 ⟨*Buffer a character* 45⟩≡ 46a ▷

```
buffer_char:
    SUBROUTINE

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

.not_0D:
    CMP     #$20
    BCC     .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     KBD_INPUT_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 53a, 54c, 85, 86, 147b, 149b, and 150c.
  **buffer_char_set_buffer_end**, used in chunk 44.
Uses **BUFF_END** 168, **KBD_INPUT_AREA** 168, **WNDWDTH** 167, and **dump_buffer_with_more** 42.

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.

46a ⟨*Buffer a character* 45⟩+≡ ◁45 46b▷

```
      LDA      #$A0  ; normal space

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

  .endloop:
```

Uses KBD_INPUT_AREA 168 and WNDWDTH 167.

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.

46b ⟨*Buffer a character* 45⟩+≡ ◁46a 47a▷

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

Uses BUFF_END 168, BUFF_LINE_LEN 168, and dump_buffer_with_more 42.

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.

47a ⟨*Buffer a character* 45⟩+≡ ◁46b

```
.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     KBD_INPUT_AREA,X
    LDX     BUFF_END
    STA     KBD_INPUT_AREA,X
    INC     BUFF_END
    LDX     BUFF_LINE_LEN
    JMP     .increment_length
```

Uses BUFF_END 168, BUFF_LINE_LEN 168, KBD_INPUT_AREA 168, and WNDWDTH 167.

### 6.1.2 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.

47b ⟨*Get alphabet* 47b⟩≡

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

.remove_shift:
    LDY     #$FF
    STY     SHIFT_ALPHABET

    get_alphabet_return:
    RTS
```

Defines:
get_alphabet, used in chunks 50a and 51.
Uses LOCKED_ALPHABET 168 and SHIFT_ALPHABET 168.

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.

48 ⟨*Get next zchar* 48⟩≡

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

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

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

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

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

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

Defines:
  get_next_zchar, used in chunks 50a, 52, and 55a.
Uses SCRATCH2 168, ZCHARS_H 168, ZCHARS_L 168, and get_next_code_word 34a.

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.

49 ⟨*Print zstring* 49⟩≡ 50a ▷

```
    print_zstring:
        SUBROUTINE

        LDA      #$00
        STA      LOCKED_ALPHABET
        STA      ZDECODE_STATE
        LDA      #$FF
        STA      SHIFT_ALPHABET
```

Defines:
  print_zstring, used in chunks 52, 55b, 115, and 124b.
Uses LOCKED_ALPHABET 168, SHIFT_ALPHABET 168, and ZDECODE_STATE 168.

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.

50a ⟨*Print zstring* 49⟩+≡ ◁49 53a▷

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

  .not_end:
      STA      SCRATCH3
      BEQ      .space
      CMP      #$01
      BEQ      .abbreviation
      CMP      #$04
      BCC      .shift_alphabet
      CMP      #$06
      BCC      .shift_lock_alphabet
      JSR      get_alphabet
```

Uses `SCRATCH3` 168, `get_alphabet` 47b, and `get_next_zchar` 48.

50b ⟨*Printing a space* 50b⟩≡

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

51 ⟨*Shifting alphabets* 51⟩≡

```
.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
```

Uses `A_mod_3` 84, `LOCKED_ALPHABET` 168, `SCRATCH3` 168, `SHIFT_ALPHABET` 168, and `get_alphabet` 47b.

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.

52 ⟨*Printing an abbreviation* 52⟩≡

```
.abbreviation:
    JSR     get_next_zchar
    ASL     A
    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     ZDECODE_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     ZDECODE_STATE
        PLA
        STA     LOCKED_ALPHABET
        LDA     #$FF            ; Resets any temporary shift
        STA     SHIFT_ALPHABET
        JMP     .loop
```

Uses LOCKED_ALPHABET 168, SCRATCH2 168, SHIFT_ALPHABET 168, ZCHARS_H 168, ZCHARS_L 168, ZCODE_PAGE_VALID2 168, ZDECODE_STATE 168, Z_ABBREV_TABLE 168, Z_PC2_H 168, Z_PC2_HH 168, Z_PC2_L 168, get_next_zchar 48, load_packed_address 35, and print_zstring 49.

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`.

53a ⟨*Print zstring* 49⟩+≡ ◁50a 53b▷

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

    .add_ascii_offset:
        CLC
        ADC     SCRATCH3

    .printchar:
        JSR     buffer_char
        JMP     .loop

    .check_for_alphabet_A1:
```

Uses SCRATCH3 168 and buffer_char 45.

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

53b ⟨*Print zstring* 49⟩+≡ ◁53a 54b▷

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

    .map_ascii_for_A2:
```

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.

54a ⟨*A2 table* 54a⟩≡

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

Defines:
`a2_table`, used in chunk 54b.

54b ⟨*Print zstring* 49⟩+≡ ◁53b

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

Uses `SCRATCH3` 168 and `a2_table` 54a.

54c ⟨*Printing a CRLF* 54c⟩≡

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

Uses `buffer_char` 45.

55a ⟨*Printing a 10-bit ZSCII character* 55a⟩≡

```
.z10bits:
    JSR     get_next_zchar
    ASL     A
    ASL     A
    ASL     A
    ASL     A
    ASL     A
    PHA
    JSR     get_next_zchar
    STA     SCRATCH3
    PLA
    ORA     SCRATCH3
    JMP     .printchar
```

Uses `SCRATCH3` 168 and `get_next_zchar` 48.

`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`.

55b ⟨*Printing a string literal* 55b⟩≡ (171)

```
print_string_literal:
    SUBROUTINE

    LDA     Z_PC
    STA     Z_PC2_L
    LDA     Z_PC+1
    STA     Z_PC2_H
    LDA     Z_PC+2
    STA     Z_PC2_HH
    LDA     #$00
    STA     ZCODE_PAGE_VALID2
    JSR     print_zstring
    LDA     Z_PC2_L
    STA     Z_PC
    LDA     Z_PC2_H
    STA     Z_PC+1
    LDA     Z_PC2_HH
    STA     Z_PC+2
    LDA     ZCODE_PAGE_VALID2
    STA     ZCODE_PAGE_VALID
    LDA     ZCODE_PAGE_ADDR2
    STA     ZCODE_PAGE_ADDR
    LDA     ZCODE_PAGE_ADDR2+1
    STA     ZCODE_PAGE_ADDR+1
    RTS
```

Uses `ZCODE_PAGE_ADDR` 168, `ZCODE_PAGE_ADDR2` 168, `ZCODE_PAGE_VALID` 168, `ZCODE_PAGE_VALID2` 168, `Z_PC` 168, `Z_PC2_H` 168, `Z_PC2_HH` 168, `Z_PC2_L` 168, and `print_zstring` 49.

### 6.1.3 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 KBD_INPUT_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.

56 ⟨*Read line* 56⟩≡

```
read_line:
    SUBROUTINE

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

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

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

```
  .loop:
      LDA      KBD_INPUT_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 58.

Uses BUFF_END 168, CURR_LINE 168, GETLN1 167, KBD_INPUT_AREA 168, OPERAND0 168, WNDTOP 167, dump_buffer_line 41, and dump_buffer_to_printer 40.

### 6.1.4 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.

58 ⟨*Instruction sread* 58⟩≡ (171) 59a ▷

```
instr_sread:
    SUBROUTINE

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

Defines:
`instr_sread`, used in chunk 87.

Uses `ADDW` 9a, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH3` 168, and `read_line` 56.

Loop:

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

59a ⟨*Instruction sread* 58⟩+≡ (171) ◁58 59b▷

```
  .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 168 and do_instruction 90.

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

59b ⟨*Instruction sread* 58⟩+≡ (171) ◁59a 59c▷

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

Uses SCRATCH3 168 and do_instruction 90.

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

59c ⟨*Instruction sread* 58⟩+≡ (171) ◁59b 60a▷

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

Uses SCRATCH3 168 and skip_separators 64.

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

60a ⟨*Instruction sread* 58⟩+≡ (171) ◁59c 60b▷

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

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

Uses SCRATCH3 168 and ZCHAR_SCRATCH1 168.

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.

60b ⟨*Instruction sread* 58⟩+≡ (171) ◁60a 61a▷

```
      LDA       INPUT_PTR           ; parsebuf[TOKEN_INDEX+3] = INPUT_PTR
      LDY       TOKEN_INDEX
      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 168, OPERAND1 168, SCRATCH3 168, is_dict_separator 65, and is_std_separator 65.

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.

61a ⟨*Instruction sread* 58⟩+≡ (171) ◁60b 61b▷

```
  .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 168, SCRATCH3 168, ZCHAR_SCRATCH1 168, and is_separator 65.

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.

61b ⟨*Instruction sread* 58⟩+≡ (171) ◁61a 62▷

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

Uses SCRATCH3 168, ZCHAR_SCRATCH1 168, and is_dict_separator 65.

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.

62 ⟨*Instruction sread* 58⟩+≡ (171) ◁61b 63▷

```
.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 168 and SCRATCH3 168.

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.

63 ⟨*Instruction sread* 58⟩+≡ (171) ◃62

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

        INY                             ; TOKEN_IDX += 4
        INY
        INY
        STY     TOKEN_IDX

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

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

Uses `OPERAND1` 168, `SCRATCH1` 168, `SCRATCH3` 168, `ascii_to_zchar` 66, and `match_dictionary_word` 73.

### Separators

64 ⟨*Skip separators* 64⟩≡ (171)

```
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 59c.

Uses `OPERAND0` 168, `SCRATCH3` 168, and `is_separator` 65.

65 ⟨*Separator checks 65*⟩≡ (171)

```
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     .found
    INY
    DEX
    BNE     .loop

.not_found:
    CLC
    RTS

.found:
    SEC
    RTS

is_dict_separator:
    SUBROUTINE

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

.loop:
    BEQ     .not_found
    INY
    CMP     (SCRATCH2),Y
    BEQ     .found
    DEX
    JMP     .loop
```

Defines:

SEPARATORS_TABLE, never used.
is_dict_separator, used in chunks 60b and 61b.
is_separator, used in chunks 61a and 64.
is_std_separator, used in chunk 60b.
Uses SCRATCH2 168 and get_dictionary_addr 72.

### 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.

66 ⟨*ASCII to Zchar* 66⟩≡ 67a ▷

```
ascii_to_zchar:
    SUBROUTINE

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

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

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

Defines:
ascii_to_zchar, used in chunk 63.
Uses LOCKED_ALPHABET 168, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, and ZCHAR_SCRATCH2 168.

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.

67a ⟨*ASCII to Zchar* 66⟩+≡ ◁66 67b▷

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

Uses SCRATCH2 168, SCRATCH3 168, and ZCHAR_SCRATCH1 168.

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.

67b ⟨*ASCII to Zchar* 66⟩+≡ ◁67a 67c▷

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

Uses LOCKED_ALPHABET 168, SCRATCH1 168, and SCRATCH3 168.

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.

67c ⟨*ASCII to Zchar* 66⟩+≡ ◁67b 68a▷

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

Uses SCRATCH1 168, SCRATCH2 168, and ZCHAR_SCRATCH1 168.

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.

68a ⟨*ASCII to Zchar* 66⟩+≡ ◁67c 68b▷
```
    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` 84, `LOCKED_ALPHABET` 168, `MOVB` 6a, and `SCRATCH1` 168.

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

68b ⟨*ASCII to Zchar* 66⟩+≡ ◁68a 68c▷
```
    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` 168 and `ZCHAR_SCRATCH2` 168.

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`.

68c ⟨*ASCII to Zchar* 66⟩+≡ ◁68b 69a▷
```
    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` 168 and `SCRATCH3` 168.

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.

69a ⟨*ASCII to Zchar* 66⟩+≡ ◁68c 69b▷

```
.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 84, LOCKED_ALPHABET 168, SCRATCH1 168, SCRATCH3 168, and ZCHAR_SCRATCH2 168.

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

69b ⟨*ASCII to Zchar* 66⟩+≡ ◁69a 70a▷

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

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

Uses SCRATCH1 168 and SCRATCH3 168.

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.

70a ⟨*ASCII to Zchar* 66⟩+≡ ◁69b 70b▷

```
  .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 168, SCRATCH3 168, and ZCHAR_SCRATCH2 168.

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.

70b ⟨*ASCII to Zchar* 66⟩+≡ ◁70a 70c▷

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

Uses SCRATCH3 168.

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.

70c ⟨*ASCII to Zchar* 66⟩+≡ ◁70b 71▷

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

Uses SCRATCH3 168.

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.

71 ⟨*ASCII to Zchar* 66⟩+≡ ◁70c

```
    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     A
    LSR     A
    LSR     A
    LSR     A
    LSR     A
    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 168, SCRATCH3 168, and ZCHAR_SCRATCH2 168.

### 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`.

72 ⟨*Get dictionary address* 72⟩≡ (171)

```
get_dictionary_addr:
    SUBROUTINE

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

Defines:
  `get_dictionary_addr`, used in chunks 65 and 73.
Uses `ADDW` 9a and `SCRATCH2` 168.

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.

73 ⟨*Match dictionary word 73*⟩≡ (171) 74a ▷

```
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     A                       ; search_size = entry length x 16
    ASL     A
    ASL     A
    ASL     A
    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 63.

Uses ADDA 7d, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, and get_dictionary_addr 72.

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.

74a ⟨*Match dictionary word* 73⟩+≡ (171) ◁73 74b▷

```
  .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 8c, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, and ZCHAR_SCRATCH2 168.

74b ⟨*Match dictionary word* 73⟩+≡ (171) ◁74a 75a▷

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

Uses ADDB2 8c, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, and SUBB2 10a.

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.

75a ⟨*Match dictionary word* 73⟩+≡ (171) ◁74b 75b▷

```
.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 8c, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, SUBB 9c, and ZCHAR_SCRATCH2 168.

If the search failed, we return 0 in SCRATCH1.

75b ⟨*Match dictionary word* 73⟩+≡ (171) ◁75a 76▷

```
.not_found:
    STOW    #$0000, SCRATCH1
    RTS
```

Uses SCRATCH1 168 and STOW 5.

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

76 ⟨*Match dictionary word* 73⟩+≡ (171) ◁75b

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

Uses SCRATCH1 168, SCRATCH2 168, and SUBW 10b.

# Chapter 7

# Arithmetic routines

### 7.0.1 Negation and sign manipulation

`negate` negates the word in `SCRATCH2`.

77a ⟨*negate* 77a⟩≡ (171)

```
  negate:
      SUBROUTINE

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

Defines:
  `negate`, used in chunks 77b, 78b, and 86.
Uses `SCRATCH2` 168 and `SUBW` 10b.

`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`.

77b ⟨*Flip sign* 77b⟩≡ (171)

```
  flip_sign:
      SUBROUTINE

      ORA       #$00
      BMI       .do_negate
      RTS

  .do_negate:
      INC       SIGN_BIT
      JMP       negate
```

Defines:
  `flip_sign`, used in chunk 78a.
Uses `negate` 77a.

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.

78a ⟨*Check sign* 78a⟩≡ (171)

```
check_sign:
    SUBROUTINE

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

Defines:
check_sign, used in chunks 136–38.
Uses SCRATCH1 168, SCRATCH2 168, and flip_sign 77b.

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.

78b ⟨*Set sign* 78b⟩≡ (171)

```
set_sign:
    SUBROUTINE

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

Defines:
set_sign, used in chunk 138.
Uses negate 77a.

### 7.0.2 16-bit multiplication

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

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

79 ⟨*mulu16* 79⟩≡ (171)

```
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 138.

Uses `ADDWC` 9b, `MOVW` 6b, `PSHW` 6c, `PULW` 7b, `RORW` 11, `SCRATCH1` 168, `SCRATCH2` 168, `SCRATCH3` 168, and `STOW` 5.

### 7.0.3 16-bit division

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

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

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

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

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

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

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

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

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

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

Here's a trace of the algorithm:

81 ⟨*trace of divu16* 81⟩≡

```
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.

82 ⟨*divu16* 82⟩≡ (171)

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

Defines:
divu16, used in chunks 85, 136, 137, and 139a.
Uses MOVW 6b, PSHW 6c, PULW 7b, ROLW 10c, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, and STOW 5.

### 7.0.4 16-bit comparison

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

83a ⟨*cmpu16* 83a⟩≡ (171)

```
cmpu16:
    SUBROUTINE

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

Defines:
`cmpu16`, used in chunks 83b and 146a.
Uses `SCRATCH1` 168 and `SCRATCH2` 168.

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

83b ⟨*cmp16* 83b⟩≡ (171)

```
cmp16:
    SUBROUTINE

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

Defines:
`cmp16`, used in chunks 142a, 144a, and 145a.
Uses `SCRATCH1` 168, `SCRATCH2` 168, and `cmpu16` 83a.

### 7.0.5 Other routines

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

84 ⟨*A mod 3* 84⟩≡

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

.end:
    RTS
```

Defines:
A_mod_3, used in chunks 51, 68a, and 69a.

### 7.0.6 Printing numbers

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

85 ⟨*Print number* 85⟩≡ (171)

```
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 chunk 151a.

Uses SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, STOB 6a, STOW 5, buffer_char 45, divu16 82, and print_negative_num 86.

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.

86 ⟨*Print negative number* 86⟩≡ (171)

```
print_negative_num:
    SUBROUTINE

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

Defines:
  print_negative_num, used in chunk 85.
Uses buffer_char 45 and negate 77a.

# Chapter 8

# The instruction dispatcher

## 8.1 Executing an instruction

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

87 ⟨*Instruction tables* 87⟩≡ (171)

```
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, never used.
routines_table_1op, used in chunk 92.
routines_table_2op, never used.

routines_table_var, used in chunk 97.
Uses illegal_opcode 123, instr_add 135b, instr_and 140a, instr_call 103, instr_clear_attr 152, instr_dec 135a, instr_dec_chk 141b, instr_div 136, instr_get_next_prop 154, instr_get_parent 155a, instr_get_prop 155b, instr_get_prop_addr 158, instr_get_prop_len 159, instr_get_sibling 160, instr_inc 134c, instr_inc_chk 142a, instr_insert_obj 161, instr_je 142b, instr_jg 144a, instr_jin 144b, instr_jl 145a, instr_jump 147a, instr_jz 145b, instr_load 130a, instr_loadb 131a, instr_loadw 130b, instr_mod 137, instr_mul 138, instr_new_line 149b, instr_nop 164a, instr_not 140b, instr_or 141a, instr_pop 133b, instr_print 150a, instr_print_addr 150b, instr_print_char 150c, instr_print_num 151a, instr_print_obj 151b, instr_print_paddr 151c, instr_print_ret 147b, instr_pull 134a, instr_push 134b, instr_put_prop 162, instr_quit 165, instr_random 139a, instr_remove_obj 163a, instr_restart 164b, instr_ret 107, instr_ret_popped 148a, instr_rfalse 148b, instr_rtrue 149a, instr_set_attr 163b, instr_sread 58, instr_store 131b, instr_storeb 133a, instr_storew 132, instr_sub 139b, instr_test 146a, and instr_test_attr 146b.

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.

89 ⟨*Execute instruction* 89⟩≡

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

Defines:
.opcode_table_jump, never used.
Uses SCRATCH1 168 and SCRATCH2 168.

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.

## 8.2 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`.

90 ⟨*Do instruction* 90⟩≡ (171) 91a ▷

```
do_instruction:
    SUBROUTINE

    LDA     Z_PC                    ; Save PC for debugging
    STA     TMP_Z_PC
    LDA     Z_PC+1
    STA     TMP_Z_PC+1
    LDA     Z_PC+2
    STA     TMP_Z_PC+2
    LDA     #$00
    STA     OPERAND_COUNT
    JSR     get_next_code_byte
    STA     CURR_OPCODE
```

Defines:
`do_instruction`, used in chunks 26b, 59, 106b, 124, 126b, 129, 131–35, 149–52, and 161–64.
Uses `CURR_OPCODE` 168, `OPERAND_COUNT` 168, `TMP_Z_PC` 168, `Z_PC` 168, and `get_next_code_byte` 28.

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

## 8.3 Decoding the instruction

Next, we determine what kind of instruction it is based on its number of operands.

91a ⟨*Do instruction* 90⟩+≡ (171) ◁90

```
    CMP     #$80
    BCS     .is_gte_80
    JMP     .do_2op
.is_gte_80:
    CMP     #$B0
    BCS     .is_gte_B0
    JMP     .do_1op
.is_gte_B0:
    CMP     #$C0
    BCC     .do_0op
    JSR     get_next_code_byte

⟨Get variable instruction operands 96⟩
```

Uses get_next_code_byte 28.

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 0op instruction table into SCRATCH2, leaving the A register with the offset into the table of the instruction to execute.

91b ⟨*Handle 0op instruction* 91b⟩≡

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

.load_opcode_table:
    PHA
    LDA     #<routine_table_0op
    STA     SCRATCH2
    LDA     #>routine_table_0op
    STA     SCRATCH2+1
    PLA
    JMP     .opcode_table_jump
```

Uses SCRATCH2 168 and illegal_opcode 123.

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, opcodes 90-9F take an 8-bit operand zero-extended to 16 bits, and opcodes A0-AF take a variable operand, whose content is 16 bits.

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

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

92 ⟨*Handle 1op instructions* 92⟩≡

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

.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

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

.1op_arg_loaded:
    LDA     #$01
    STA     OPERAND_COUNT
    LDA     SCRATCH2
    STA     OPERAND0
    LDA     SCRATCH2+1
    STA     OPERAND0+1
    LDA     CURR_OPCODE
    AND     #$0F
    CMP     #$10
    BCC     .go_to_1op
    JMP     illegal_opcode

.go_to_1op:
    PHA
    LDA     #<routines_table_1op
    STA     SCRATCH2
    LDA     #>routines_table_1op
    STA     SCRATCH2+1
    PLA
    JMP     .opcode_table_jump
```

Uses CURR_OPCODE 168, OPERAND0 168, OPERAND_COUNT 168, SCRATCH2 168, get_const_byte 98a, get_const_word 98b, get_var_content 99, illegal_opcode 123, and routines_table_1op 87.

Handling a 2op-type instruction (opcodes `00-7F`) is a little more complicated. Opcodes `00-3F` are followed by a single byte to be zero-extended into a 16-bit operand, while opcodes `40-7F` are followed by a single byte representing a variable address. After that, opcodes `00-1F` and `40-5F` have a single byte to be zero-extended into a 16-bit operand, and the other opcodes are followed by a single byte representing a variable address.

Operand are stored consecutively in `OPERAND0-OPERAND1`. `OPERAND_COUNT` is set to 2.

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

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

94 ⟨*Handle 2op instruction* 94⟩≡

```
.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:
    LDA      SCRATCH2
    STA      OPERAND0
    LDA      SCRATCH2+1
    STA      OPERAND0+1
    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:
    LDA      SCRATCH2
    STA      OPERAND1
    LDA      SCRATCH2+1
    STA      OPERAND03
    LDA      #$02
    STA      OPERAND_COUNT
    LDA      CURR_OPCODE

.check_for_good_2op:
```

| 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 |

```
    AND     #$1F
    CMP     #$19
    BCC     .go_to_op2
    JMP     illegal_opcode

.go_to_op2:
    PHA
    LDA     #<routine_table_2op
    STA     SCRATCH2
    LDA     #>routine_table_2op
    STA     SCRATCH2+1
    PLA
    JMP     .opcode_table_jump
```

Defines:
`.check_for_good_2op`, never used.

Uses `CURR_OPCODE` 168, `OPERAND0` 168, `OPERAND1` 168, `OPERAND_COUNT` 168, `SCRATCH2` 168, `get_const_byte` 98a, `get_var_content` 99, and `illegal_opcode` 123.

Handling a varop-type instruction (opcodes `C0-FF`) is the most complicated. 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`.

96 ⟨*Get variable instruction operands* 96⟩≡ (91a)

```
    LDX     #$00

.get_next_operand:
    PHA
    TAY
    TXA
    PHA
    TYA
    AND     #$C0
    BNE     .is_01_10_11
    JSR     get_const_word            ; handle 00
    JMP     .store_operand

.is_01_10_11:
    CMP     #$80
    BNE     .is_10_11
    JSR     get_var_content           ; handle 01
    JMP     .store_operand

.is_10_11:
    CMP     #$40
    BNE     .is_11
    JSR     get_const_byte            ; handle 10
    JMP     .store_operand

.is_11:
    PLA
    PLA
    JMP     .handle_varoperand_opcode ; handle 11 (ends operand list)

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

```
    SEC
    ROL       A
    SEC
    ROL       A
    JMP       .get_next_operand

⟨Handle var operand opcode 97⟩
```

Uses `OPERAND0` 168, `OPERAND_COUNT` 168, `SCRATCH2` 168, `get_const_byte` 98a, `get_const_word` 98b, and `get_var_content` 99.

Once the operands are loaded, we load `SCRATCH2` with the address of the table containing the instruction addresses. Opcodes `E0-FF` are VAR-type instructions, although opcodes `EA-FF` are illegal in version 1 of the interpreter. Opcodes `C0-DF` are 2op-type instructions, although opcodes `D9-DF` are illegal.

97 ⟨*Handle var operand opcode* 97⟩≡ (96)

```
.handle_varoperand_opcode:
    LDA       #<routines_table_var
    STA       SCRATCH2
    LDA       #>routines_table_var
    STA       SCRATCH2+1
    LDA       CURR_OPCODE
    CMP       #$E0
    BCS       .is_vararg_instr
    JMP       .check_for_good_2op

.is_vararg_instr:
    SBC       #$E0
    CMP       #$0A
    BCC       .opcode_table_jump
    JMP       illegal_opcode
```

Uses `CURR_OPCODE` 168, `SCRATCH2` 168, `illegal_opcode` 123, and `routines_table_var` 87.

## 8.4 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`.

98a ⟨*Get const byte* 98a⟩≡ (171)

```
get_const_byte:
    SUBROUTINE

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

Defines:
`get_const_byte`, used in chunks 92, 94, and 96.
Uses `SCRATCH2` 168 and `get_next_code_byte` 28.

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.

98b ⟨*Get const word* 98b⟩≡ (171)

```
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 92 and 96.
Uses `SCRATCH2` 168 and `get_next_code_byte` 28.

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`.

99 ⟨*Get var content* 99⟩≡ (171)

```
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     A
    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     A
    STA     SCRATCH1
    LDA     #$00
    ROL     A
    STA     SCRATCH1+1

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

```
    ADC     SCRATCH1
    STA     SCRATCH1
    LDA     Z_GLOBALVARS+1
    ADC     SCRATCH1+1
    STA     SCRATCH1+1

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

get_top_of_stack:
    SUBROUTINE

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

Defines:
get_nonstack_var, used in chunk 100.
get_top_of_stack, never used.
get_var_content, used in chunks 92, 94, and 96.
Uses GLOBAL_ZVARS_ADDR 168, LOCAL_ZVARS 168, SCRATCH1 168, SCRATCH2 168, Z_PC 168, and get_next_code_byte 28.

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

100 ⟨*Get var content in A* 100⟩≡ (171)

```
var_get:
    SUBROUTINE

    ORA     #$00
    BEQ     pop_push
    JMP     get_nonstack_var
```

Defines:
var_get, used in chunks 125 and 130a.
Uses get_nonstack_var 99 and pop_push 102.

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.

101 ⟨*Store var* 101⟩≡ (171)

```
store_var:
    SUBROUTINE

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

store_var2:
    SUBROUTINE

    ORA     #$00
    BNE     .nonstack
    JMP     push

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

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

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

Defines:
store_var, used in chunks 124a and 153.
Uses GLOBAL_ZVARS_ADDR 168, LOCAL_ZVARS 168, SCRATCH1 168, SCRATCH2 168, and get_next_code_byte 28.

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.

102 ⟨*Store to var A* 102⟩≡ (171)

```
    var_put:
        SUBROUTINE

        ORA     #$00
        BEQ     .pop_push
        JMP     store_var2

    pop_push:
        JSR     pop
        JMP     push

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

Defines:
pop_push, used in chunk 100.
var_put, used in chunks 125a and 131b.
Uses SCRATCH2 168.

# Chapter 9

# Calls and returns

## 9.1 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.

103 ⟨*Instruction call* 103⟩≡ (171) 104a ▷

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

Defines:
`instr_call`, used in chunk 87.

Uses `OPERAND0` 168, `SCRATCH2` 168, `STOW` 5, and `store_and_next` 124a.

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.

104a ⟨*Instruction call* 103⟩+≡ (171) ◁103 104b▷

```
  .push_frame:
      MOV     FRAME_STACK_COUNT, SCRATCH2
      MOV     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 MOVW 6b, SCRATCH2 168, STOB 6a, ZCODE_PAGE_VALID 168, and Z_PC 168.

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

104b ⟨*Instruction call* 103⟩+≡ (171) ◁104a 104c▷

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

Uses OPERAND0 168 and Z_PC 168.

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

104c ⟨*Instruction call* 103⟩+≡ (171) ◁104b 105▷

```
      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 28.

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

105 ⟨*Instruction call* 103⟩+≡ (171) ◁104c 106a▷

```
    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` 168, `SCRATCH2` 168, and `get_next_code_byte` 28.

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

106a ⟨*Instruction call* 103⟩+≡ (171) ◁105 106b▷

```
  .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, STRATCH1         ; operand = 0
      STOB    #$00, STRATCH2         ; zvar = 0

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

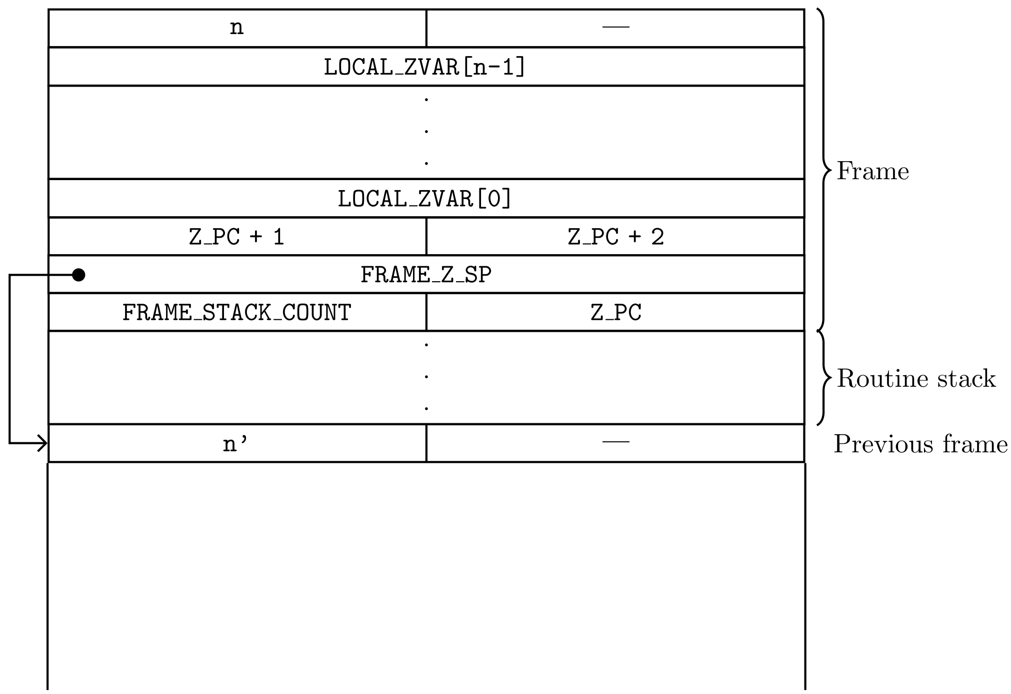
Uses LOCAL_ZVARS 168, OPERAND0 168, OPERAND_COUNT 168, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, and STOB 6a.

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.

106b ⟨*Instruction call* 103⟩+≡ (171) ◁106a

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

Uses MOVW 6b, PULB 7a, SCRATCH2 168, STACK_COUNT 168, Z_SP 168, and do_instruction 90.




## 9.2 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.

107 ⟨*Instruction ret* 107⟩≡ (171) 108a ▷

```
instr_ret:
    SUBROUTINE

    MOVW     FRAME_Z_SP, Z_SP
    MOVB     FRAME_STACK_COUNT, STACK_COUNT
```

Defines:
  `instr_ret`, used in chunks 87, 148a, and 149a.
Uses `MOVB` 6a, `MOVW` 6b, `STACK_COUNT` 168, and `Z_SP` 168.

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.

108a ⟨*Instruction ret* 107⟩+≡ (171) ◁107 108b▷

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

Uses SCRATCH2 168.

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

108b ⟨*Instruction ret* 107⟩+≡ (171) ◁108a 108c▷

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

Uses ADDA 7d, LOCAL_ZVARS 168, MOVB 6a, SCRATCH1 168, SCRATCH2 168, SCRATCH3 168, and STOW 5.

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

108c ⟨*Instruction ret* 107⟩+≡ (171) ◁108b 108d▷

```
.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 168, SCRATCH2 168, SCRATCH3 168, and SUBB 9c.

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

108d ⟨*Instruction ret* 107⟩+≡ (171) ◁108c 109▷

```
.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 MOVB 6a, MOVW 6b, SCRATCH2 168, and Z_PC 168.

Finally, we store the return value.

109 ⟨*Instruction ret* 107⟩+≡ (171) ◁108d

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

Uses MOVW 6b, OPERAND0 168, SCRATCH2 168, STOB 6a, ZCODE_PAGE_VALID 168, and store_and_next 124a.

# Chapter 10

# Objects

## 10.1 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).

## 10.2 Getting an object's address

The `get_object_address` routine gets the address of the object number in the `A` register and puts it in `SCRATCH2`.

It does this by first setting `SCRATCH2` to 9 times the `A` register (since objects entries are 9 bytes long).

110 ⟨*Get object address* 110⟩≡ (171) 111a ▷

```
get_object_addr:
    SUBROUTINE

    STA     SCRATCH2
    LDA     #$00
```

```
      STA     SCRATCH2+1
      LDA     SCRATCH2
      ASL     SCRATCH2
      ROL     SCRATCH2+1
      ASL     SCRATCH2
      ROL     SCRATCH2+1
      ASL     SCRATCH2
      ROL     SCRATCH2+1
      CLC
      ADC     SCRATCH2
      BCC     .continue
      INC     SCRATCH2+1
      CLC

  .continue:
```

Defines:
get_object_addr, used in chunks 112, 114–16, 118, 144b, 153, 155a, 160, and 161.
Uses SCRATCH2 168.

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.

111a ⟨*Get object address* 110⟩+≡ (171) ◁110 111b▷

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

  .continue2:
```

Uses SCRATCH2 168.

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.

111b ⟨*Get object address* 110⟩+≡ (171) ◁111a

```
      LDY     #HEADER_OBJECT_TABLE_ADDR_OFFSET-1
      LDA     (Z_HEADER_ADDR),Y
      CLC
      ADC     SCRATCH2
      STA     SCRATCH2
      DEY
      LDA     (Z_HEADER_ADDR),Y
      ADC     SCRATCH2+1
      ADC     Z_HEADER_ADDR+1
      STA     SCRATCH2+1
      RTS
```

Uses HEADER_OBJECT_TABLE_ADDR_OFFSET 170 and SCRATCH2 168.

## 10.3 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.

112a ⟨*Remove object* 112a⟩≡

```
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 161 and 163a.
Uses OBJECT_PARENT_OFFSET 170, OPERAND0 168, SCRATCH2 168, and get_object_addr 110.

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

112b ⟨*Remove obj* 112b⟩≡ (171) 112c ▷

```
    TAX                              ; save obj_ptr
    LDA      SCRATCH2
    PHA
    LDA      SCRATCH2+1
    PHA
    TXA
```

Uses SCRATCH2 168.

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

112c ⟨*Remove obj* 112b⟩+≡ (171) ◁ 112b 113a ▷

```
    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 170, SCRATCH2 168, and get_object_addr 110.

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.

113a ⟨*Remove obj* 112b⟩+≡ (171) ◁112c 113b▷

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

Uses OPERAND0 168.

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

113b ⟨*Remove obj* 112b⟩+≡ (171) ◁113a 114a▷

```
        PLA                             ; restore obj_ptr
        STA     SCRATCH1+1
        PLA
        STA     SCRATCH1
        LDA     SCRATCH1
        PHA
        LDA     SCRATCH1+1
        PHA
        LDY     #OBJECT_SIBLING_OFFSET ; A = obj_ptr->next
        LDA     (SCRATCH1),Y
        LDY     #OBJECT_CHILD_OFFSET   ; parent_ptr->child = A
        STA     (SCRATCH2),Y
        JMP     .detach
```

Uses OBJECT_CHILD_OFFSET 170, OBJECT_SIBLING_OFFSET 170, SCRATCH1 168, and SCRATCH2 168.

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

113c ⟨*Detach obj* 113c⟩≡ (114b)

```
    .detach:
        PLA                             ; restore obj_ptr
        STA     SCRATCH2+1
        PLA
        STA     SCRATCH2
        LDY     #OBJECT_PARENT_OFFSET  ; obj_ptr->parent = 0
        LDA     #$00
        STA     (SCRATCH2),Y
        INY
        STA     (SCRATCH2),Y
        RTS
```

Uses OBJECT_PARENT_OFFSET 170 and SCRATCH2 168.

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.

114a ⟨*Remove obj* 112b⟩+≡ (171) ◁113b 114b▷

```
.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` 170, `OPERAND0` 168, `SCRATCH2` 168, and `get_object_addr` 110.

`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.

114b ⟨*Remove obj* 112b⟩+≡ (171) ◁114a

```
    PLA                            ; restore obj_ptr
    STA     SCRATCH1+1
    PLA
    STA     SCRATCH1
    LDA     SCRATCH1
    PHA
    LDA     SCRATCH1+1
    PHA
    LDA     (SCRATCH1),Y           ; child_ptr->next = obj_ptr->next
    STA     (SCRATCH2),Y

  ⟨Detach obj 113c⟩
```

Uses `SCRATCH1` 168 and `SCRATCH2` 168.

## 10.4 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.

115 ⟨*Print object in A* 115⟩≡ (171)

```
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 chunk 151b.

Uses INCW 7c, MOVW 6b, OBJECT_PROPS_OFFSET 170, SCRATCH1 168, SCRATCH2 168, get_object_addr 110, load_address 34b, and print_zstring 49.

## 10.5 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.

116 ⟨*Get attribute pointer and mask* 116⟩≡ (171) 117a ▷

```
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 146b, 152, and 163b.
Uses INCW 7c, OPERAND0 168, OPERAND1 168, SCRATCH1 168, SCRATCH2 168, and get_object_addr 110.

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`.

117a ⟨*Get attribute pointer and mask* 116⟩+≡ (171) ◁116 117b▷

```
    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` 168, `SCRATCH3` 168, and `STOW` 5.

Finally, we load the attribute word into `SCRATCH1`.

117b ⟨*Get attribute pointer and mask* 116⟩+≡ (171) ◁117a

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

Uses `SCRATCH1` 168 and `SCRATCH2` 168.

## 10.6 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.

118 ⟨*Get property pointer* 118⟩≡ (171)

```
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     A
    TAY
    INY
    RTS
```

Defines:
`get_property_ptr`, used in chunks 154, 155b, 158, and 162.

Uses `ADDW` 9a, `OBJECT_PROPS_OFFSET` 170, `OPERAND0` 168, `SCRATCH1` 168, `SCRATCH2` 168, and `get_object_addr` 110.

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

119a ⟨*Get property number* 119a⟩≡ (171)
```
get_property_num:
    SUBROUTINE

    LDA      (SCRATCH2),Y
    AND      #$1F
    RTS
```
Defines:
get_property_num, used in chunks 154, 155b, 158, and 162.
Uses SCRATCH2 168.

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

119b ⟨*Get property length* 119b⟩≡ (171)
```
get_property_len:
    SUBROUTINE

    LDA      (SCRATCH2),Y
    ROR      A
    ROR      A
    ROR      A
    ROR      A
    ROR      A
    AND      #$07
    RTS
```
Defines:
get_property_len, used in chunks 120, 157, 159, and 162.
Uses SCRATCH2 168.

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

120 ⟨*Next property* 120⟩≡ (171)
```
next_property:
    SUBROUTINE

    JSR     get_property_len
    TAX

.loop:
    INY
    DEX
    BPL     .loop
    INY
    RTS
```
Defines:
`next_property`, used in chunks 154, 155b, 158, and 162.
Uses `get_property_len` 119b.

# Chapter 11

# 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 |

## 11.1 Instruction utilities

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

illegal_opcode hits a BRK instruction.

123 ⟨*Instruction illegal opcode* 123⟩≡ (171)

```
instr_illegal_opcode:
    SUBROUTINE

    JSR      brk
```

Defines:
illegal_opcode, used in chunks 87, 91b, 92, 94, and 97.
Uses brk 24c.

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.

124a ⟨*Store and go to next instruction* 124a⟩≡ (171)

```
store_zero_and_next:
    SUBROUTINE

    LDA     #$00

store_A_and_next:
    SUBROUTINE

    STA     SCRATCH2
    LDA     #$00
    STA     SCRATCH2+1

store_and_next:
    SUBROUTINE

    JSR     store_var
    JMP     do_instruction
```

Defines:
  store_A_and_next, used in chunks 154 and 159.
  store_and_next, used in chunks 103, 109, 130, 131a, 135b, 137–41, and 155–58.
  store_zero_and_next, used in chunks 154 and 158.
Uses SCRATCH2 168, do_instruction 90, and store_var 101.

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

124b ⟨*Print zstring and go to next instruction* 124b⟩≡ (171)

```
print_zstring_and_next:
    SUBROUTINE

    JSR     print_zstring
    JMP     do_instruction
```

Defines:
  print_zstring_and_next, used in chunks 150b and 151c.
Uses do_instruction 90 and print_zstring 49.

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

125a ⟨*Increment variable* 125a⟩≡ (171)

```
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 134c and 142a.
Uses INCW 7c, OPERAND0 168, PSHW 6c, PULW 7b, SCRATCH2 168, var_get 100, and var_put 102.

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

125b ⟨*Decrement variable* 125b⟩≡ (171)

```
dec_var:
    SUBROUTINE

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

Defines:
dec_var, used in chunks 135a and 141b.
Uses OPERAND0 168, SCRATCH2 168, SUBB 9c, and var_get 100.

### 11.1.1 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.

126a ⟨*Handle branch* 126a⟩≡ (171) 126b ▷

```
negated_branch:
    SUBROUTINE

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

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

Defines:
`branch`, used in chunks 142, 143, and 145b.
`negated_branch`, used in chunks 142–46 and 153.
Uses `get_next_code_byte` 28.

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.

126b ⟨*Handle branch* 126a⟩+≡ (171) ◁ 126a 127 ▷

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

.next:
    JMP     do_instruction
```

Uses `do_instruction` 90 and `get_next_code_byte` 28.

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.

127 ⟨*Handle branch* 126a⟩+≡ (171) ◁126b 128a▷

```
.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 168 and get_next_code_byte 28.

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.

128a ⟨*Handle branch* 126a⟩+≡ (171) ◁127 128b▷

```
  .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 168, instr_rfalse 148b, and instr_rtrue 149a.

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.

128b ⟨*Handle branch* 126a⟩+≡ (171) ◁128a 128c▷

```
  branch_to_offset:
      SUBROUTINE

      SUBB      SCRATCH2, #$01
```

Defines:
branch_to_offset, used in chunk 147a.
Uses SCRATCH2 168 and SUBB 9c.

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

128c ⟨*Handle branch* 126a⟩+≡ (171) ◁128b 129▷

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

Uses SCRATCH1 168 and SCRATCH2 168.

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.

129 ⟨*Handle branch* 126a⟩+≡ (171) ◁128c

```
    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 168, SCRATCH2 168, ZCODE_PAGE_VALID 168, Z_PC 168, and do_instruction 90.

## 11.2 Data movement instructions

### 11.2.1 load

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

130a ⟨*Instruction load* 130a⟩≡ (171)

```
instr_load:
    SUBROUTINE

    LDA      OPERAND0
    JSR      var_get
    JMP      store_and_next
```

Defines:
  `instr_load`, used in chunk 87.
Uses `OPERAND0` 168, `store_and_next` 124a, and `var_get` 100.

### 11.2.2 loadw

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

130b ⟨*Instruction loadw* 130b⟩≡ (171)

```
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 87.
Uses `ADDW` 9a, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH2` 168, `get_next_code_word` 34a, `load_address` 34b, and `store_and_next` 124a.

### 11.2.3 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.

131a ⟨*Instruction loadb* 131a⟩≡ (171)

```
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 87.

Uses `ADDW` 9a, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH2` 168, `get_next_code_byte2` 32, `load_address` 34b, and `store_and_next` 124a.

### 11.2.4 store

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

131b ⟨*Instruction store* 131b⟩≡ (171)

```
instr_store:
    SUBROUTINE

    MOVW     OPERAND1, SCRATCH2
    LDA      OPERAND0

stretch_var_put:
    JSR      var_put
    JMP      do_instruction
```

Defines:
`instr_store`, used in chunk 87.
`stretch_var_put`, used in chunk 134a.

Uses `MOVW` 6b, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH2` 168, `do_instruction` 90, and `var_put` 102.

### 11.2.5 storew

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

132 ⟨*Instruction storew* 132⟩≡ (171)

```
instr_storew:
    SUBROUTINE

    LDA     OPERAND1        ; SCRATCH2 = Z_HEADER_ADDR + OPERAND0 + 2*OPERAND1
    ASL     A
    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 87.

Uses `ADDW` 9a, `OPERAND0` 168, `OPERAND1` 168, `OPERAND2` 168, `SCRATCH2` 168, and `do_instruction` 90.

### 11.2.6 storeb

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

133a ⟨*Instruction storeb* 133a⟩≡ (171)

```
  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 87.
Uses `ADDW` 9a, `OPERAND0` 168, `OPERAND1` 168, `OPERAND2` 168, `SCRATCH2` 168, and `do_instruction` 90.

## 11.3 Stack instructions

### 11.3.1 pop

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

133b ⟨*Instruction pop* 133b⟩≡ (171)

```
  instr_pop:
      SUBROUTINE

      JSR      pop
      JMP      do_instruction
```

Defines:
  `instr_pop`, used in chunk 87.
Uses `do_instruction` 90.

### 11.3.2 pull

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

134a ⟨*Instruction pull* 134a⟩≡ (171)

```
  instr_pull:
      SUBROUTINE

      JSR       pop
      LDA       OPERAND0
      JMP       stretch_var_put
```

Defines:
`instr_pull`, used in chunk 87.
Uses `OPERAND0` 168 and `stretch_var_put` 131b.

### 11.3.3 push

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

134b ⟨*Instruction push* 134b⟩≡ (171)

```
  instr_push:
      SUBROUTINE

      MOVW      OPERAND0, SCRATCH2
      JSR       push
      JMP       do_instruction
```

Defines:
`instr_push`, used in chunk 87.
Uses `MOVW` 6b, `OPERAND0` 168, `SCRATCH2` 168, and `do_instruction` 90.

## 11.4 Decrements and increments

### 11.4.1 inc

`inc` increments the variable in the operand.

134c ⟨*Instruction inc* 134c⟩≡ (171)

```
  instr_inc:
      SUBROUTINE

      JSR       inc_var
      JMP       do_instruction
```

Defines:
`instr_inc`, used in chunk 87.
Uses `do_instruction` 90 and `inc_var` 125a.

### 11.4.2 dec

`dec` decrements the variable in the operand.

135a ⟨*Instruction dec* 135a⟩≡ (171)

```
instr_dec:
    SUBROUTINE

    JSR     dec_var
    JMP     do_instruction
```

Defines:
`instr_dec`, used in chunk 87.
Uses `dec_var` 125b and `do_instruction` 90.

## 11.5 Arithmetic instructions

### 11.5.1 add

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

135b ⟨*Instruction add* 135b⟩≡ (171)

```
instr_add:
    SUBROUTINE

    ADDW    OPERAND0, OPERAND1, SCRATCH2
    JMP     store_and_next
```

Defines:
`instr_add`, used in chunk 87.
Uses `ADDW` 9a, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH2` 168, and `store_and_next` 124a.

### 11.5.2 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.

136 ⟨*Instruction div* 136⟩≡ (171)

```
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 87.

Uses `MOVW` 6b, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH1` 168, `SCRATCH2` 168, `check_sign` 78a, and `divu16` 82.

### 11.5.3 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.

137 ⟨*Instruction mod* 137⟩≡ (171)

```
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 87.

Uses `MOVW` 6b, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH1` 168, `SCRATCH2` 168, `check_sign` 78a, `divu16` 82, and `store_and_next` 124a.

### 11.5.4 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.

138 ⟨*Instruction mul* 138⟩≡ (171)

```
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 87.

Uses MOVW 6b, OPERAND0 168, OPERAND1 168, SCRATCH1 168, SCRATCH2 168, check_sign 78a, mulu16 79, set_sign 78b, and store_and_next 124a.

### 11.5.5 random

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

139a ⟨*Instruction random* 139a⟩≡ (171)

```
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 87.

Uses `INCW` 7c, `MOVW` 6b, `OPERAND0` 168, `SCRATCH1` 168, `SCRATCH2` 168, `divu16` 82, and `store_and_next` 124a.

### 11.5.6 sub

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

139b ⟨*Instruction sub* 139b⟩≡ (171)

```
instr_sub:
    SUBROUTINE

    SUBW    OPERAND1, OPERAND0, SCRATCH2
    JMP     store_and_next
```

Defines:
`instr_sub`, used in chunk 87.

Uses `OPERAND0` 168, `OPERAND1` 168, `SCRATCH2` 168, `SUBW` 10b, and `store_and_next` 124a.

## 11.6 Logical instructions

### 11.6.1 and

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

140a ⟨*Instruction and* 140a⟩≡ (171)

```
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 87.
Uses `OPERAND0` 168, `OPERAND1` 168, `SCRATCH2` 168, and `store_and_next` 124a.

### 11.6.2 not

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

140b ⟨*Instruction not* 140b⟩≡ (171)

```
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 87.
Uses `OPERAND0` 168, `SCRATCH2` 168, and `store_and_next` 124a.

### 11.6.3 or

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

141a ⟨*Instruction or* 141a⟩≡ (171)

```
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 87.
Uses OPERAND0 168, OPERAND1 168, SCRATCH2 168, and store_and_next 124a.

## 11.7 Conditional branch instructions

### 11.7.1 dec_chk

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

141b ⟨*Instruction dec chk* 141b⟩≡ (171)

```
instr_dec_chk:
    SUBROUTINE

    JSR     dec_var
    MOVW    OPERAND1, SCRATCH1
    JMP     do_chk
```

Defines:
instr_dec_chk, used in chunk 87.
Uses MOVW 6b, OPERAND1 168, SCRATCH1 168, dec_var 125b, and do_chk 142a.

### 11.7.2 inc_chk

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

142a ⟨*Instruction inc chk* 142a⟩≡ (171)

```
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 141b.
`instr_inc_chk`, used in chunk 87.
`stretch_to_branch`, used in chunks 144–46.

Uses `MOVW` 6b, `OPERAND1` 168, `SCRATCH1` 168, `SCRATCH2` 168, `branch` 126a, `cmp16` 83b, `inc_var` 125a, and `negated_branch` 126a.

### 11.7.3 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`.

142b ⟨*Instruction je* 142b⟩≡ (171) 143a ▷

```
instr_je:
    SUBROUTINE

    LDX     OPERAND_COUNT
    DEX
    BNE     .check_second
    JSR     brk
```

Defines:
`instr_je`, used in chunk 87.

Uses `OPERAND_COUNT` 168 and `brk` 24c.

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

143a ⟨*Instruction je* 142b⟩+≡ (171) ◁142b 143b▷

```
  .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` 168, `OPERAND1` 168, and `branch` 126a.

Next we do the same with the third operand.

143b ⟨*Instruction je* 142b⟩+≡ (171) ◁143a 143c▷

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

  .check_next2:
      DEX
      BEQ       .neg_branch
```

Uses `OPERAND0` 168 and `branch` 126a.

And again with the fourth operand.

143c ⟨*Instruction je* 142b⟩+≡ (171) ◁143b

```
      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` 168, `branch` 126a, and `negated_branch` 126a.

### 11.7.4 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.

144a ⟨*Instruction jg* 144a⟩≡ (171)
```
instr_jg:
    SUBROUTINE

    MOVW      OPERAND0, SCRATCH1
    MOVW      OPERAND1, SCRATCH2
    JSR       cmp16
    BCC       stretch_to_branch
    JMP       negated_branch
```
Defines:
`instr_jg`, used in chunk 87.

Uses `MOVW` 6b, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH1` 168, `SCRATCH2` 168, `cmp16` 83b, `negated_branch` 126a, and `stretch_to_branch` 142a.

### 11.7.5 jin

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

144b ⟨*Instruction jin* 144b⟩≡ (171)
```
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 87.

Uses `OBJECT_PARENT_OFFSET` 170, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH2` 168, `get_object_addr` 110, `negated_branch` 126a, and `stretch_to_branch` 142a.

### 11.7.6 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.

145a ⟨*Instruction jl* 145a⟩≡ (171)
```
instr_jl:
    SUBROUTINE

    MOVW     OPERAND0, SCRATCH2
    MOVW     OPERAND1, SCRATCH1
    JSR      cmp16
    BCC      stretch_to_branch
    JMP      negated_branch
```
Defines:
`instr_jl`, used in chunk 87.

Uses `MOVW` 6b, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH1` 168, `SCRATCH2` 168, `cmp16` 83b, `negated_branch` 126a, and `stretch_to_branch` 142a.

### 11.7.7 jz

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

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

145b ⟨*Instruction jz* 145b⟩≡ (171)
```
instr_jz:
    SUBROUTINE

    LDA      OPERAND0+1
    ORA      OPERAND0
    BEQ      take_branch
    JMP      negated_branch

take_branch:
    JMP      branch
```
Defines:
`instr_jz`, used in chunk 87.
`take_branch`, used in chunk 153.

Uses `OPERAND0` 168, `branch` 126a, and `negated_branch` 126a.

### 11.7.8 test

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

146a ⟨*Instruction test* 146a⟩≡ (171)

```
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 87.

Uses `MOVB` 6a, `OPERAND0` 168, `OPERAND1` 168, `SCRATCH1` 168, `SCRATCH2` 168, `cmpu16` 83a, `negated_branch` 126a, and `stretch_to_branch` 142a.

### 11.7.9 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.

146b ⟨*Instruction test attr* 146b⟩≡ (171)

```
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 87.

Uses `SCRATCH1` 168, `SCRATCH3` 168, `attr_ptr_and_mask` 116, `negated_branch` 126a, and `stretch_to_branch` 142a.

## 11.8 Jump and subroutine instructions

### 11.8.1 call

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

### 11.8.2 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.

147a ⟨*Instruction jump* 147a⟩≡ (171)

```
instr_jump:
    SUBROUTINE

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

Defines:
`instr_jump`, used in chunk 87.
Uses `MOVW` 6b, `OPERAND0` 168, `SCRATCH2` 168, `SUBB` 9c, and `branch_to_offset` 128b.

### 11.8.3 print_ret

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

147b ⟨*Instruction print ret* 147b⟩≡ (171)

```
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 87.
Uses `buffer_char` 45 and `instr_rtrue` 149a.

### 11.8.4 ret

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

### 11.8.5 ret_popped

ret_popped pops the stack and returns that value.

148a ⟨*Instruction ret popped* 148a⟩≡ (171)

```
  instr_ret_popped:
      SUBROUTINE

      JSR     pop
      MOVW    SCRATCH2, OPERAND0
      JMP     instr_ret
```

Defines:
  instr_ret_popped, used in chunk 87.
Uses MOVW 6b, OPERAND0 168, SCRATCH2 168, and instr_ret 107.

### 11.8.6 rfalse

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

148b ⟨*Instruction rfalse* 148b⟩≡ (171)

```
  instr_rfalse:
      SUBROUTINE

      LDA     #$00
      JMP     ret_a
```

Defines:
  instr_rfalse, used in chunks 87 and 128a.
Uses ret_a 149a.

### 11.8.7 rtrue

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

149a ⟨*Instruction rtrue* 149a⟩≡ (171)
```
instr_rtrue:
    SUBROUTINE

    LDA     #$01
ret_a:
    STA     OPERAND0
    LDA     #$00
    STA     OPERAND0+1
    JMP     instr_ret
```
Defines:
instr_rtrue, used in chunks 87, 128a, and 147b.
ret_a, used in chunk 148b.
Uses OPERAND0 168 and instr_ret 107.

## 11.9 Print instructions

### 11.9.1 new_line

new_line prints CRLF.

149b ⟨*Instruction new line* 149b⟩≡ (171)
```
instr_new_line:
    SUBROUTINE

    LDA     #$0D
    JSR     buffer_char
    LDA     #$0A
    JSR     buffer_char
    JMP     do_instruction
```
Defines:
instr_new_line, used in chunk 87.
Uses buffer_char 45 and do_instruction 90.

### 11.9.2 print

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

150a ⟨*Instruction print* 150a⟩≡ (171)
```
instr_print:
    SUBROUTINE

    JSR     print_literal_string
    JMP     do_instruction
```
Defines:
  `instr_print`, used in chunk 87.
Uses `do_instruction` 90.

### 11.9.3 print_addr

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

150b ⟨*Instruction print addr* 150b⟩≡ (171)
```
instr_print_addr:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2
    JSR     load_address
    JMP     print_zstring_and_next
```
Defines:
  `instr_print_addr`, used in chunk 87.
Uses `MOVW` 6b, `OPERAND0` 168, `SCRATCH2` 168, `load_address` 34b, and `print_zstring_and_next` 124b.

### 11.9.4 print_char

`print_char` prints the one-byte ASCII character in `OPERAND0`.

150c ⟨*Instruction print char* 150c⟩≡ (171)
```
instr_print_char:
    SUBROUTINE

    LDA     OPERAND0
    JSR     buffer_char
    JMP     do_instruction
```
Defines:
  `instr_print_char`, used in chunk 87.
Uses `OPERAND0` 168, `buffer_char` 45, and `do_instruction` 90.

### 11.9.5 print_num

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

151a ⟨*Instruction print num* 151a⟩≡ (171)

```
instr_print_num:
    SUBROUTINE

    MOVW    OPERAND0, SCRATCH2
    JSR     print_number
    JMP     do_instruction
```

Defines:
instr_print_num, used in chunk 87.
Uses MOVW 6b, OPERAND0 168, SCRATCH2 168, do_instruction 90, and print_number 85.

### 11.9.6 print_obj

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

151b ⟨*Instruction print obj* 151b⟩≡ (171)

```
instr_print_obj:
    SUBROUTINE

    LDA     OPERAND0
    JSR     print_obj_in_A
    JMP     do_instruction
```

Defines:
instr_print_obj, used in chunk 87.
Uses OPERAND0 168, do_instruction 90, and print_obj_in_A 115.

### 11.9.7 print_paddr

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

151c ⟨*Instruction print paddr* 151c⟩≡ (171)

```
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 87.
Uses MOVW 6b, OPERAND0 168, SCRATCH2 168, load_packed_address 35,
and print_zstring_and_next 124b.

## 11.10 Object instructions

### 11.10.1 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.

152 ⟨*Instruction clear attr* 152⟩≡ (171)

```
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 87.

Uses `SCRATCH1` 168, `SCRATCH2` 168, `SCRATCH3` 168, `attr_ptr_and_mask` 116, and `do_instruction` 90.

### 11.10.2 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`).

153 ⟨*Instruction get child* 153⟩≡ (171)

```
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 160.

Uses `OBJECT_CHILD_OFFSET` 170, `OPERAND0` 168, `SCRATCH2` 168, `get_object_addr` 110, `negated_branch` 126a, `store_var` 101, and `take_branch` 145b.

### 11.10.3 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.

154 ⟨*Instruction get next prop* 154⟩≡ (171)

```
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 87.

Uses OPERAND1 168, get_property_num 119a, get_property_ptr 118, next_property 120, store_A_and_next 124a, and store_zero_and_next 124a.

### 11.10.4 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.

155a ⟨*Instruction get parent* 155a⟩≡ (171)

```
instr_get_parent:
    SUBROUTINE

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

Defines:
instr_get_parent, used in chunk 87.

Uses OBJECT_PARENT_OFFSET 170, OPERAND0 168, SCRATCH2 168, get_object_addr 110, and store_and_next 124a.

### 11.10.5 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.

155b ⟨*Instruction get prop* 155b⟩≡ (171) 156▷

```
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 87.

Uses `OPERAND1` 168, `get_property_num` 119a, `get_property_ptr` 118, and `next_property` 120.

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.

156 ⟨*Instruction get prop* 155b⟩+≡ (171) ◁155b 157▷

```
.get_default:
    LDY      #HEADER_OBJECT_TABLE_ADDR_OFFSET
    CLC
    LDA      (Z_HEADER_ADDR),Y
    ADC      Z_HEADER_ADDR
    STA      SCRATCH1
    DEY
    LDA      (Z_HEADER_ADDR),Y
    ADC      Z_HEADER_ADDR+1
    STA      SCRATCH1+1              ; table_ptr
    LDA      OPERAND1                ; SCRATCH2 <- table_ptr[2*OPERAND1]
    ASL      A
    TAY
    DEY
    LDA      (SCRATCH1),Y
    STA      SCRATCH2
    DEY
    LDA      (SCRATCH1),Y
    STA      SCRATCH2+1
    JMP      store_and_next
```

Uses HEADER_OBJECT_TABLE_ADDR_OFFSET 170, OPERAND1 168, SCRATCH1 168, SCRATCH2 168, and store_and_next 124a.

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.

157 ⟨*Instruction get prop* 155b⟩+≡ (171) ◁156

```
.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` 6b, `SCRATCH1` 168, `SCRATCH2` 168, `brk` 24c, `get_property_len` 119b, and `store_and_next` 124a.

### 11.10.6 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.

158 ⟨*Instruction get prop addr* 158⟩≡ (171)

```
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 87.

Uses ADDAC 8a, INCW 7c, OPERAND1 168, SCRATCH2 168, SUBW 10b, get_property_num 119a, get_property_ptr 118, next_property 120, store_and_next 124a, and store_zero_and_next 124a.

### 11.10.7 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.

159 ⟨*Instruction get prop len* 159⟩≡ (171)

```
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 87.

Uses OPERAND0 168, SCRATCH2 168, get_property_len 119b, and store_A_and_next 124a.

### 11.10.8 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`).

160 ⟨*Instruction get sibling* 160⟩≡ (171)

```
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 87.
Uses `OBJECT_SIBLING_OFFSET` 170, `OPERAND0` 168, `get_object_addr` 110, and `push_and_check_obj` 153.

### 11.10.9 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.

161 ⟨*Instruction insert obj* 161⟩≡ (171)

```
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 87.

Uses OBJECT_CHILD_OFFSET 170, OBJECT_PARENT_OFFSET 170, OBJECT_SIBLING_OFFSET 170, OPERAND0 168, OPERAND1 168, PSHW 6c, PULW 7b, SCRATCH2 168, do_instruction 90, get_object_addr 110, and remove_obj 112a.

### 11.10.10 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.

162 ⟨*Instruction put prop* 162⟩≡ (171)

```
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 87.

Uses OPERAND1 168, OPERAND2 168, SCRATCH2 168, brk 24c, do_instruction 90, get_property_len 119b, get_property_num 119a, get_property_ptr 118, and next_property 120.

### 11.10.11 remove_obj

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

163a ⟨*Instruction remove obj* 163a⟩≡ (171)

```
instr_remove_obj:
    SUBROUTINE

    JSR     remove_obj
    JMP     do_instruction
```

Defines:
`instr_remove_obj`, used in chunk 87.
Uses `do_instruction` 90 and `remove_obj` 112a.

### 11.10.12 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.

163b ⟨*Instruction set attr* 163b⟩≡ (171)

```
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 87.
Uses `SCRATCH1` 168, `SCRATCH2` 168, `SCRATCH3` 168, `attr_ptr_and_mask` 116, and `do_instruction` 90.

## 11.11 Other instructions

### 11.11.1 nop

`nop` does nothing.

164a ⟨*Instruction nop* 164a⟩≡ (171)

```
instr_nop:
    SUBROUTINE

    JMP      do_instruction
```

Defines:
`instr_nop`, used in chunk 87.
Uses `do_instruction` 90.

### 11.11.2 restart

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

164b ⟨*Instruction restart* 164b⟩≡

```
instr_restart:
    SUBROUTINE

    JSR      dump_buffer_with_more
    JMP      main
```

Defines:
`instr_restart`, used in chunk 87.
Uses `dump_buffer_with_more` 42 and `main` 19a.

### 11.11.3 restore

`restore` restores the game. See the section on restoring the game.

### 11.11.4 quit

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

165 ⟨*Instruction quit* 165⟩≡

```
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 87.
Uses `SCRATCH2` 168, `STOW` 5, and `dump_buffer_with_more` 42.

### 11.11.5 save

`save` saves the game. See the section on saving the game.

### 11.11.6 sread

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

# Chapter 12

# The entire program

166a ⟨*main.asm* 166a⟩≡

```
       PROCESSOR 6502
```

⟨*defines* 166b⟩
⟨*routines* 171⟩

166b ⟨*defines* 166b⟩≡ (166a)
⟨*Apple ROM defines* 167⟩
⟨*Program defines* 168⟩
⟨*Table offsets* 170⟩

167 ⟨*Apple ROM defines* 167⟩≡ (166b)

```
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

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 chunk 42.
- `CLREOL`, used in chunk 42.
- `COUT`, used in chunks 40 and 43b.
- `COUT1`, used in chunks 36, 39, and 43a.
- `CSW`, used in chunks 40 and 43b.
- `CV`, never used.
- `GETLN1`, used in chunk 56.
- `HOME`, used in chunk 37.
- `INIT`, used in chunks 16 and 17.
- `INVFLG`, used in chunks 38 and 42.
- `IWMDATAPTR`, used in chunks 14a, 15e, and 17.
- `IWMSECTOR`, used in chunks 15c and 17.
- `IWMSLTNDX`, used in chunks 14–17.
- `PROMPT`, used in chunk 38.
- `RDKEY`, used in chunk 42.
- `RDSECT_PTR`, used in chunks 13c, 14b, and 17.
- `SETKBD`, used in chunks 16 and 17.
- `SETVID`, used in chunks 16 and 17.
- `VTAB`, never used.
- `WNDBTM`, used in chunks 38 and 42.
- `WNDLFT`, used in chunk 38.
- `WNDTOP`, used in chunks 37, 38, 42, and 56.
- `WNDWDTH`, used in chunks 38, 43a, and 45–47.

168 ⟨*Program defines* 168⟩≡ (166b)

```
CURR_OPCODE         EQU     $80
OPERAND_COUNT       EQU     $81
OPERAND0            EQU     $82     ; 2 bytes
OPERAND1            EQU     $84     ; 2 bytes
OPERAND2            EQU     $86     ; 2 bytes
OPERAND3            EQU     $88     ; 2 bytes
Z_PC                EQU     $8A     ; 3 bytes
ZCODE_PAGE_ADDR     EQU     $8D     ; 2 bytes
ZCODE_PAGE_VALID    EQU     $8F
PAGE_TABLE_INDEX    EQU     $90
Z_PC2_H             EQU     $91
Z_PC2_HH            EQU     $92
Z_PC2_L             EQU     $93
ZCODE_PAGE_ADDR2    EQU     $94     ; 2 bytes
ZCODE_PAGE_VALID2   EQU     $96
PAGE_TABLE_INDEX2   EQU     $97
GLOBAL_ZVARS_ADDR   EQU     $98     ; 2 bytes
LOCAL_ZVARS         EQU     $9A     ; 30 bytes
AFTER_Z_IMAGE_ADDR  EQU     $B8
CURR_DISK_BUFF_ADDR EQU     $BA     ; 2 bytes
NUM_IMAGE_PAGES     EQU     $BC
NUM_PAGE_TABLE_ENTRIES  EQU $BD
LAST_Z_PAGE         EQU     $BF
PAGE_L_TABLE        EQU     $C0     ; 2 bytes
PAGE_H_TABLE        EQU     $C2     ; 2 bytes
NEXT_PAGE_TABLE     EQU     $C4     ; 2 bytes
PREV_PAGE_TABLE     EQU     $C6     ; 2 bytes
STACK_COUNT         EQU     $C8
Z_SP                EQU     $C9     ; 2 bytes
SHIFT_ALPHABET      EQU     $CE
LOCKED_ALPHABET     EQU     $CF
ZDECODE_STATE       EQU     $D0
ZCHARS_L            EQU     $D1
ZCHARS_H            EQU     $D2
ZCHAR_SCRATCH1      EQU     $D3     ; 6 bytes
ZCHAR_SCRATCH2      EQU     $DA     ; 6 bytes
TOKEN_INDEX         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
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
KBD_INPUT_AREA      EQU     $0200
```

Defines:

170 ⟨*Table offsets* 170⟩≡ (166b)

```
HEADER_DICT_OFFSET       EQU      $08
HEADER_OBJECT_TABLE_ADDR_OFFSET  EQU     $0B
HEADER_FLAGS2_OFFSET     EQU      $10
FIRST_OBJECT_OFFSET      EQU      $35

OBJECT_PARENT_OFFSET     EQU      $04
OBJECT_SIBLING_OFFSET    EQU      $05
OBJECT_CHILD_OFFSET      EQU      $06
OBJECT_PROPS_OFFSET      EQU      $07
```

Defines:
- `FHEADER_FLAGS2_OFFSET`, never used.
- `HEADER_OBJECT_TABLE_ADDR_OFFSET`, used in chunks 111b and 156.
- `IRST_OBJECT_OFFSET`, never used.
- `OBJECT_CHILD_OFFSET`, used in chunks 112c, 113b, 153, and 161.
- `OBJECT_PARENT_OFFSET`, used in chunks 112a, 113c, 144b, 155a, and 161.
- `OBJECT_PROPS_OFFSET`, used in chunks 115 and 118.
- `OBJECT_SIBLING_OFFSET`, used in chunks 113b, 114a, 160, and 161.

171 ⟨*routines* 171⟩≡ (166a)

```
        ORG     $0800
```

⟨*main* 19a⟩

⟨*Instruction tables* 87⟩

⟨*Do instruction* 90⟩
⟨*Get const byte* 98a⟩
⟨*Get const word* 98b⟩
⟨*Get var content in A* 100⟩
⟨*Store to var A* 102⟩
⟨*Get var content* 99⟩
⟨*Store and go to next instruction* 124a⟩
⟨*Store var* 101⟩
⟨*Handle branch* 126a⟩
⟨*Instruction rtrue* 149a⟩
⟨*Instruction rfalse* 148b⟩
⟨*Instruction print* 150a⟩
⟨*Printing a string literal* 55b⟩
⟨*Instruction print ret* 147b⟩
⟨*Instruction nop* 164a⟩
⟨*Instruction ret popped* 148a⟩
⟨*Instruction pop* 133b⟩
⟨*Instruction new line* 149b⟩
⟨*Instruction jz* 145b⟩
⟨*Instruction get sibling* 160⟩
⟨*Instruction get child* 153⟩
⟨*Instruction get parent* 155a⟩
⟨*Instruction get prop len* 159⟩
⟨*Instruction inc* 134c⟩
⟨*Instruction dec* 135a⟩
⟨*Increment variable* 125a⟩
⟨*Decrement variable* 125b⟩
⟨*Instruction print addr* 150b⟩
⟨*Instruction illegal opcode* 123⟩
⟨*Instruction remove obj* 163a⟩
⟨*Remove obj* 112b⟩
⟨*Instruction print obj* 151b⟩
⟨*Print object in A* 115⟩
⟨*Instruction ret* 107⟩
⟨*Instruction jump* 147a⟩
⟨*Instruction print paddr* 151c⟩
⟨*Print zstring and go to next instruction* 124b⟩
⟨*Instruction load* 130a⟩
⟨*Instruction not* 140b⟩
⟨*Instruction jl* 145a⟩
⟨*Instruction jg* 144a⟩
⟨*Instruction dec chk* 141b⟩
⟨*Instruction inc chk* 142a⟩

⟨*Instruction jin* 144b⟩
⟨*Instruction test* 146a⟩
⟨*Instruction or* 141a⟩
⟨*Instruction and* 140a⟩
⟨*Instruction test attr* 146b⟩
⟨*Instruction set attr* 163b⟩
⟨*Instruction clear attr* 152⟩
⟨*Instruction store* 131b⟩
⟨*Instruction insert obj* 161⟩
⟨*Instruction loadw* 130b⟩
⟨*Instruction loadb* 131a⟩
⟨*Instruction get prop* 155b⟩
⟨*Instruction get prop addr* 158⟩
⟨*Instruction get next prop* 154⟩
⟨*Instruction add* 135b⟩
⟨*Instruction sub* 139b⟩
⟨*Instruction mul* 138⟩
⟨*Instruction div* 136⟩
⟨*Instruction mod* 137⟩
⟨*Instruction je* 142b⟩
⟨*Instruction call* 103⟩
⟨*Instruction storew* 132⟩
⟨*Instruction storeb* 133a⟩
⟨*Instruction put prop* 162⟩
⟨*Instruction sread* 58⟩
⟨*Skip separators* 64⟩
⟨*Separator checks* 65⟩
⟨*Get dictionary address* 72⟩
⟨*Match dictionary word* 73⟩
⟨*Instruction print char* 150c⟩
⟨*Instruction print num* 151a⟩
⟨*Print number* 85⟩
⟨*Print negative number* 86⟩
⟨*Instruction random* 139a⟩
⟨*Instruction push* 134b⟩
⟨*Instruction pull* 134a⟩
⟨*mulu16* 79⟩
⟨*divu16* 82⟩
⟨*Check sign* 78a⟩
⟨*Set sign* 78b⟩
⟨*negate* 77a⟩
⟨*Flip sign* 77b⟩
⟨*Get attribute pointer and mask* 116⟩
⟨*Get property pointer* 118⟩
⟨*Get property number* 119a⟩
⟨*Get property length* 119b⟩
⟨*Next property* 120⟩
⟨*Get object address* 110⟩
⟨*cmp16* 83b⟩
⟨*cmpu16* 83a⟩

# Chapter 13

# Defined Chunks

# Chapter 14

# Index