Chapter 1

Lode Runner

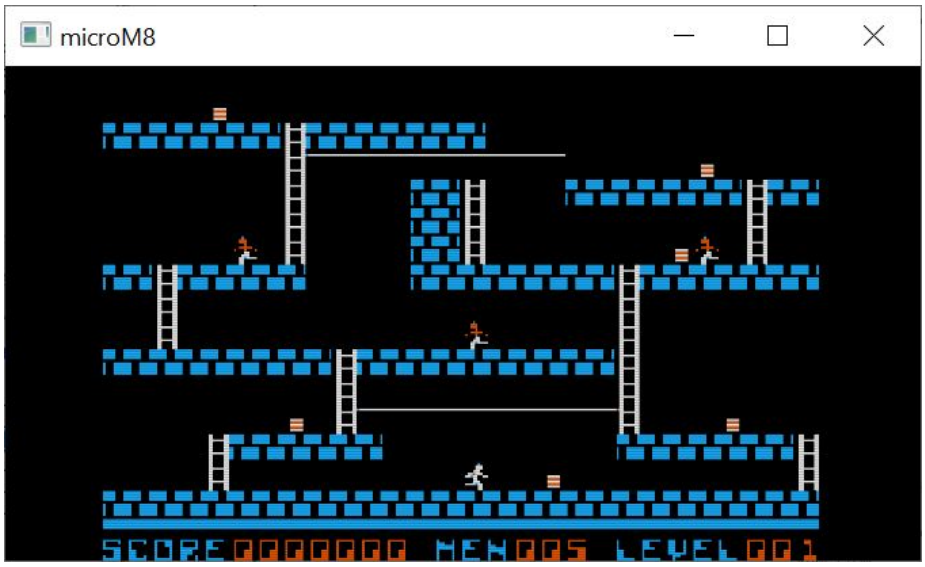
1.1 Introduction

Lode Runner was a game originally written in 1982 by Douglas E. Smith (1960–2014) for the Apple II series of computers, and published by Broderbund.

You control the movement of your character, moving left and right along brick and bedrock platforms, climbing ladders, and "monkey-traversing" ropes strung across gaps. The object is to collect all the gold boxes while avoiding being touched by the guards. You can dig holes in brick parts of the floor which can allow you to reach otherwise unreachable caverns, and the holes can also

trap the guards for a short while. Holes fill themselves in after a short time period, and if you're in a hole when that happens, you lose a life. However, if a guard is in the hole and the hole fills, the guard disappears and reappears somewhere along the top of the screen.

You get points for collecting boxes and forcing guards to respawn. Once you collect all the boxes, a ladder will appear leading out of the top of the screen. This gets you to the next level, and play continues.



Lode Runner included 150 levels and also a level editor.

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 `Lode_Runner_1983_Broderbund_cr_Reset_Vector.do` 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 Lode Runner 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 DOS

Since programs generally come on disk, and such disks are genrally bootable, the only thing the disk card does is load the data of the disk on track 0 sector 0 to location 0800 and then jump to it. Thus, any additional services need to be supplied by the disk as a disk operating system. The most popular ones were DOS 3.3 and ProDOS.

Lode Runner contains just the parts of DOS 3.3 it needs. See the section on Disk routines for more information.

2.7 Temporaries and scratch space

4 ⟨*defines* 4⟩≡ (281) 22▷

```
TMP_PTR          EQU     $0A     ; 2 bytes
TMP              EQU     $1A
SCRATCH_5C       EQU     $5C
MATH_TMPL        EQU     $6F
MATH_TMPH        EQU     $70
TMP_LOOP_CTR     EQU     $88
SCRATCH_A1       EQU     $A1
```

Defines:
MATH_TMPH, used in chunks 90, 102, and 103a.
MATH_TMPL, used in chunks 90, 102, and 103a.
SCRATCH_5C, used in chunks 63, 224, and 250.
SCRATCH_A1, used in chunks 70, 71, and 145.
TMP, used in chunks 112b, 114, 116, and 213.
TMP_LOOP_CTR, used in chunks 126 and 145.
TMP_PTR, used in chunks 5, 26, 60a, and 256.

Chapter 3

Apple II Graphics

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

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

We have routines to clear these screens.

5 ⟨*routines* 5⟩≡ (281) 26▷

```
      ORG     $7A51
  CLEAR_HGR1:
      SUBROUTINE

      LDA     #$20                ; Start at $2000
      LDX     #$40                ; End at $4000 (but not including)
      BNE     CLEAR_PAGE          ; Unconditional jump

  CLEAR_HGR2:
      SUBROUTINE

      LDA     #$40                ; Start at $4000
      LDX     #$60                ; End at $6000 (but not including)
      ; fallthrough

  CLEAR_PAGE:
      STA     TMP_PTR+1           ; Start with the page in A.
      LDA     #$00
      STA     TMP_PTR
      TAY
      LDA     #$80                ; fill byte = 0x80

  .loop:
      STA     (TMP_PTR),Y
      INY
      BNE     .loop
```

```
    INC     TMP_PTR+1
    CPX     TMP_PTR+1
    BNE     .loop               ; while TMP_PTR != X * 0x100
    RTS
```

Defines:
  CLEAR_HGR1, used in chunks 53, 124, and 259.
  CLEAR_HGR2, used in chunks 53, 119b, 145, 237, 265, and 268.
Uses TMP_PTR 4.

3.1 Pixels and their color

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

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

- Pixels are drawn to the screen from byte data least significant bit first. This means that for the first byte bit 0 is column 0, bit 1 is column 1, and so on.
- A pattern of `11` results in two white pixels at the `1` positions.
- A pattern of `010` results at least in a colored pixel at the `1` position.
- A pattern of `101` results at least in a colored pixel at the `0` position.
- So, a pattern of `01010` results in at least three consecutive colored pixels starting from the first `1` to the last `1`. The last `0` bit would also be colored if followed by a `1`.
- Likewise, a pattern of `11011` results in two white pixels, a colored pixel, and then two more white pixels.
- The color of a `010` pixel depends on the column that the `1` falls on, and also whether the high bit of its byte was set or not.
- The color of a `11011` pixel depends on the column that the `0` falls on, and also whether the high bit of its byte was set or not.

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

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

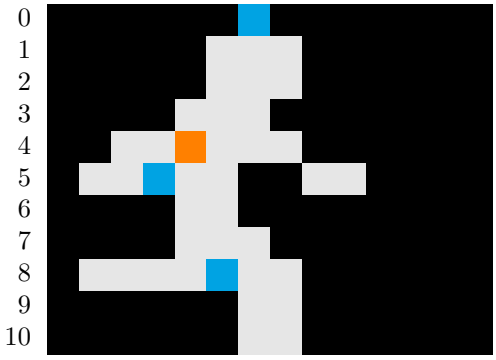
An example would probably be good here. We will take one of the sprites from the game.

Bytes	Bits	Pixel Data
00 00	0000000 0000000	00000000000000
00 00	0000000 0000000	00000000000000
00 00	0000000 0000000	00000000000000
55 00	1010101 0000000	10101010000000
41 00	1000001 0000000	10000010000000
01 00	0000001 0000000	10000000000000
55 00	1010101 0000000	10101010000000
50 00	1010000 0000000	00001010000000
50 00	1010000 0000000	00001010000000
51 00	1010001 0000000	10001010000000
55 00	1010101 0000000	10101010000000

The game automatically sets the high bit of each byte, so we know we're going to see orange and blue. Assuming that the following bits are all zero, and we place the sprite starting at column 0, we should see this:

Here is a more complex sprite:

Bytes	Bits	Pixel Data
40 00	1000000 0000000	00000010000000
60 01	1100000 0000001	00000111000000
60 01	1100000 0000001	00000111000000
70 00	1110000 0000000	00001110000000
6C 01	1101100 0000001	00110111000000
36 06	0110110 0000110	01101100110000
30 00	0110000 0000000	00001100000000
70 00	1110000 0000000	00001110000000
5E 01	1011110 0000001	01111011000000
40 01	1000000 0000001	00000011000000
40 01	1000000 0000001	00000011000000



Take note of the orange and blue pixels. All the patterns noted in the rules above are used.

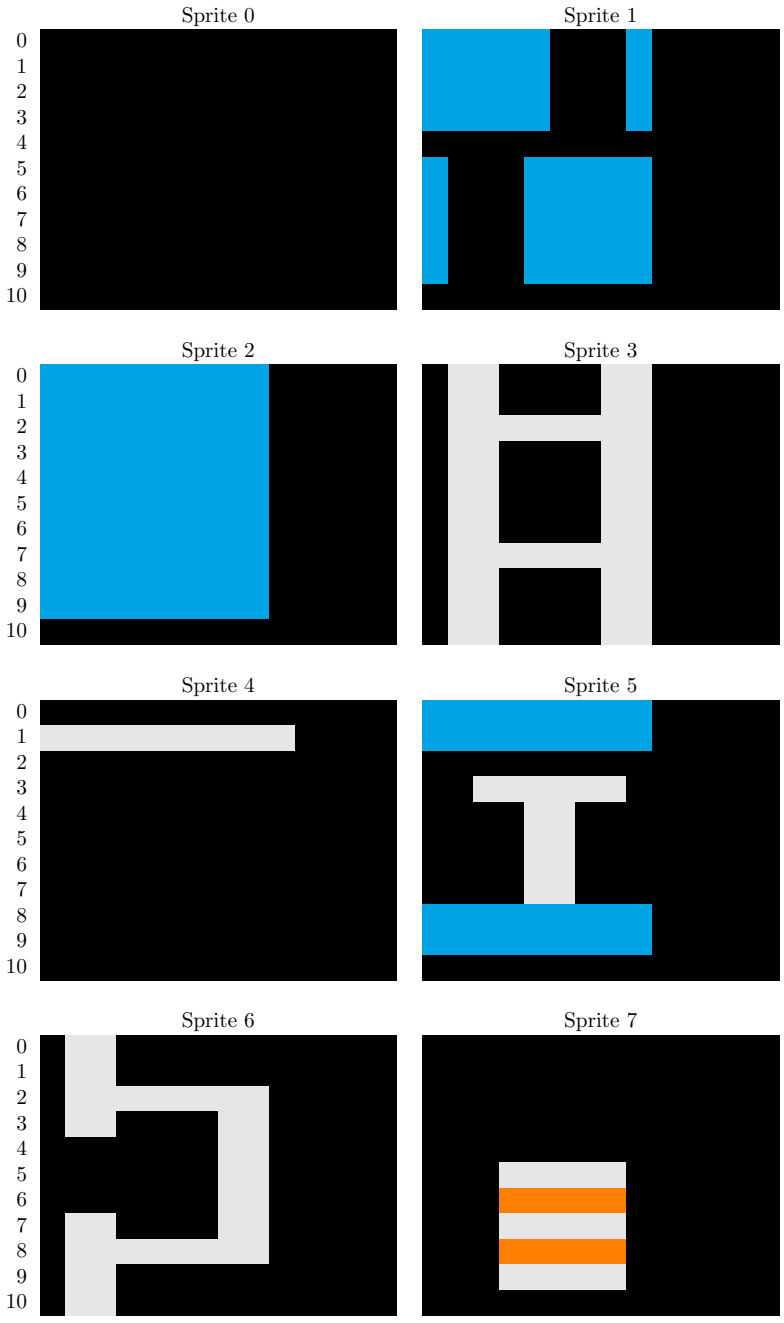
3.2 The sprites

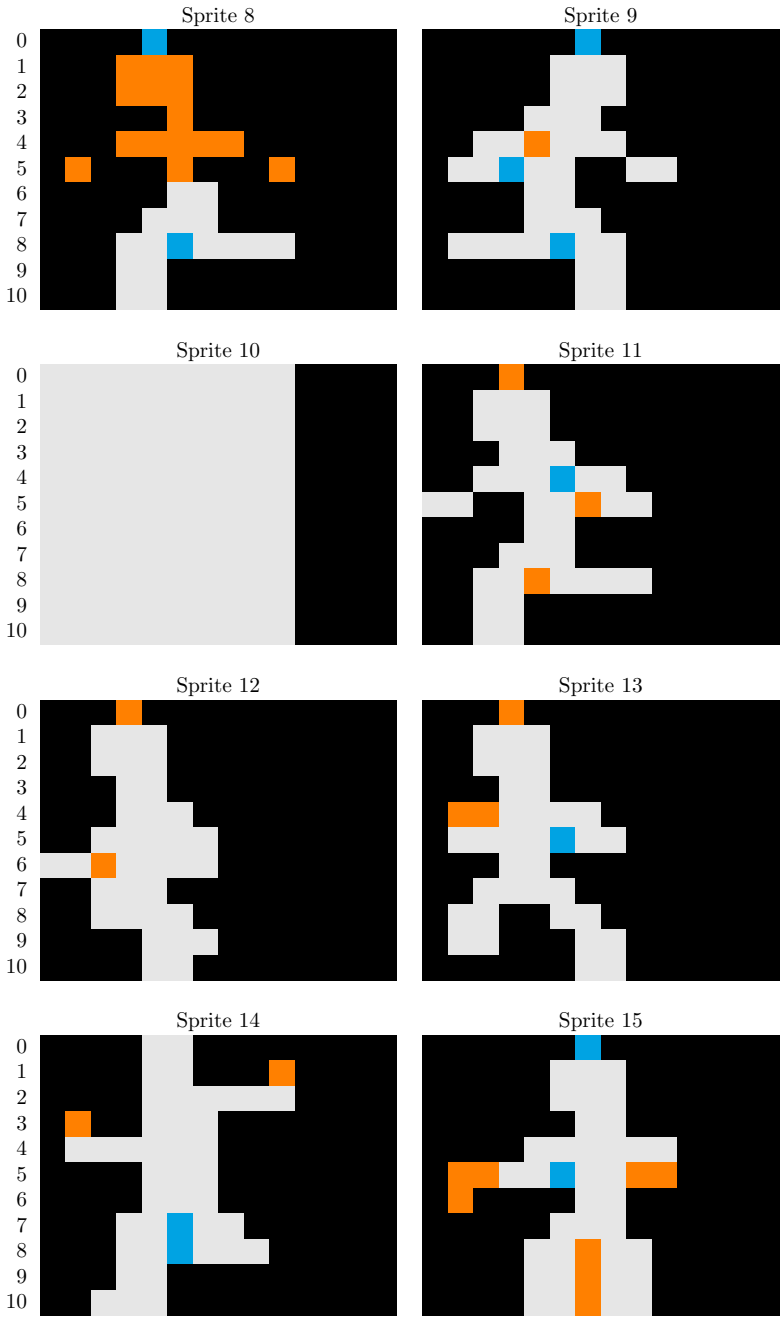
Lode Runner defines 104 sprites, each being 11 rows, with two bytes per row. The first bytes of all 104 sprites are in the table first, then the second bytes, then the third bytes, and so on. Later we will see that only the leftmost 10 pixels out of the 14-pixel description is used.

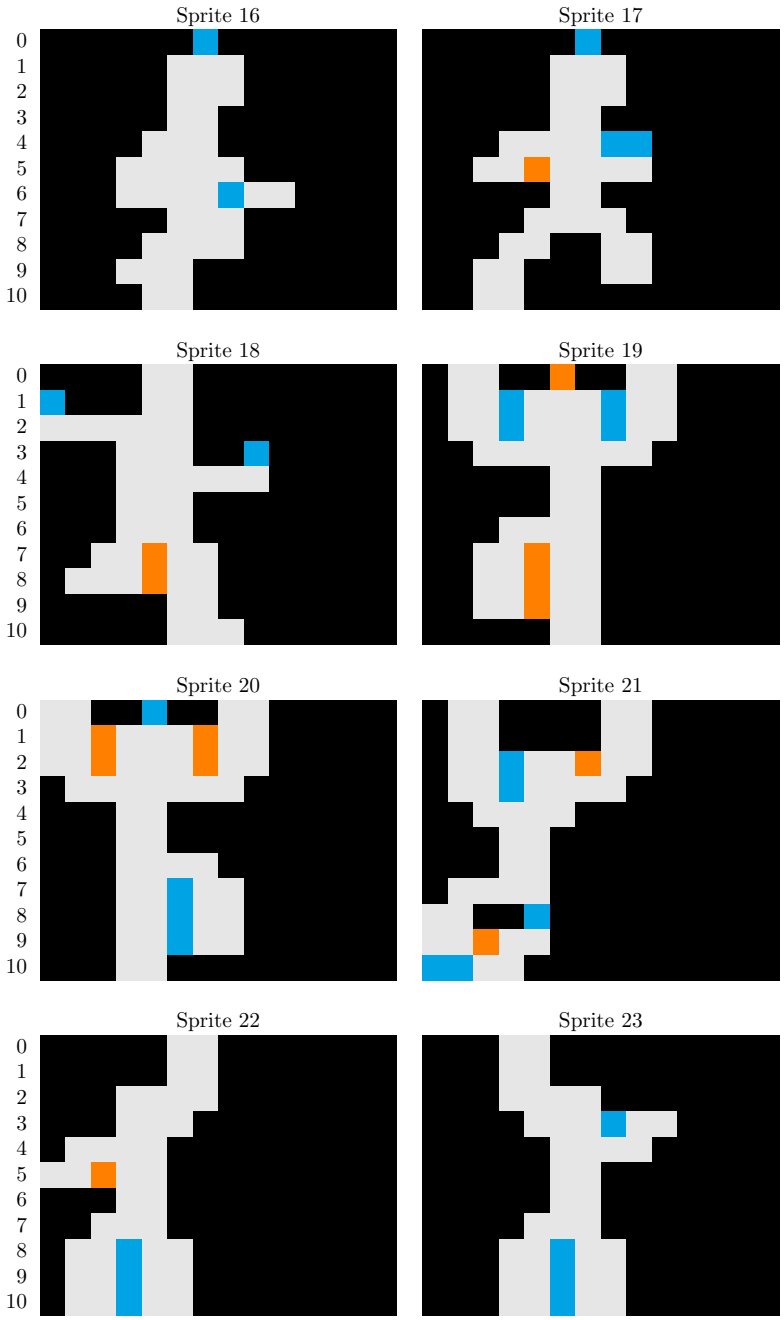
9 ⟨*tables* 9⟩≡ (281) 24▷

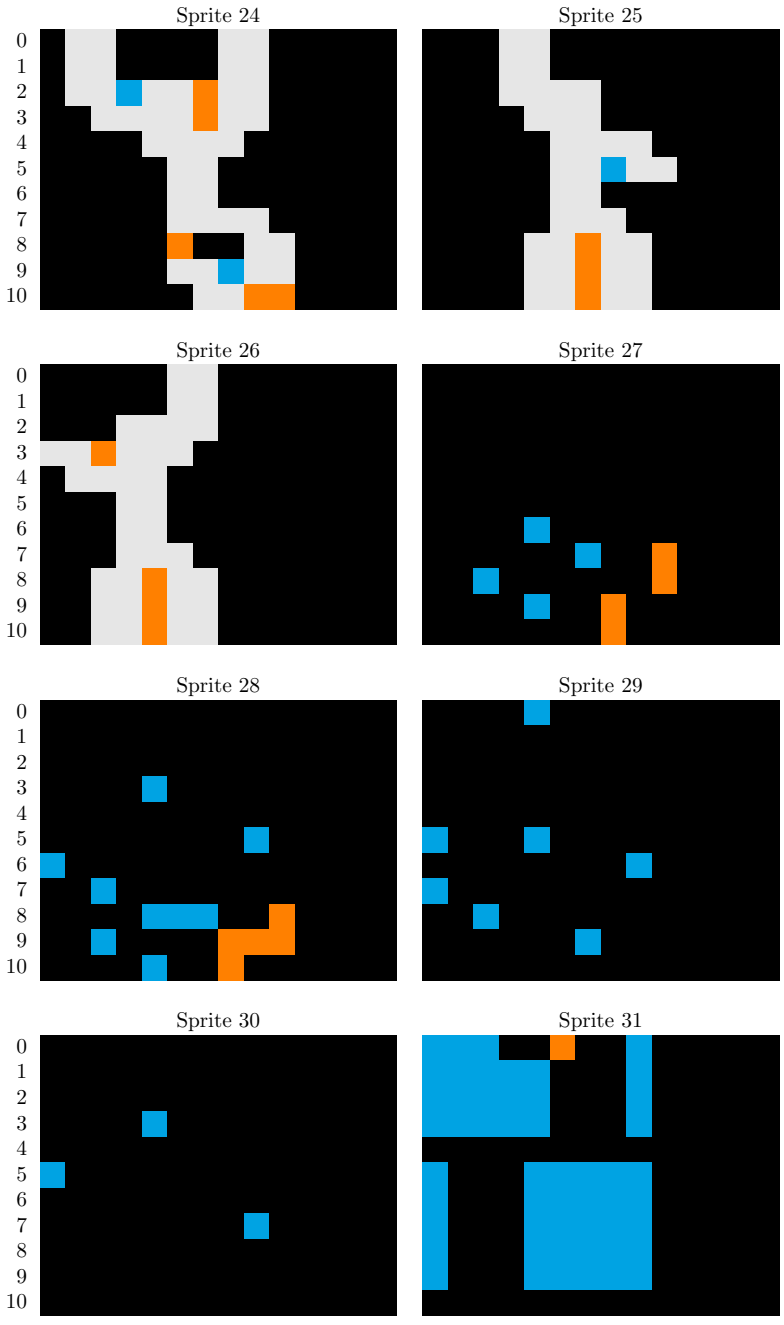
```
    ORG     $AD00
SPRITE_DATA:
    INCLUDE "sprite_data.asm"
```

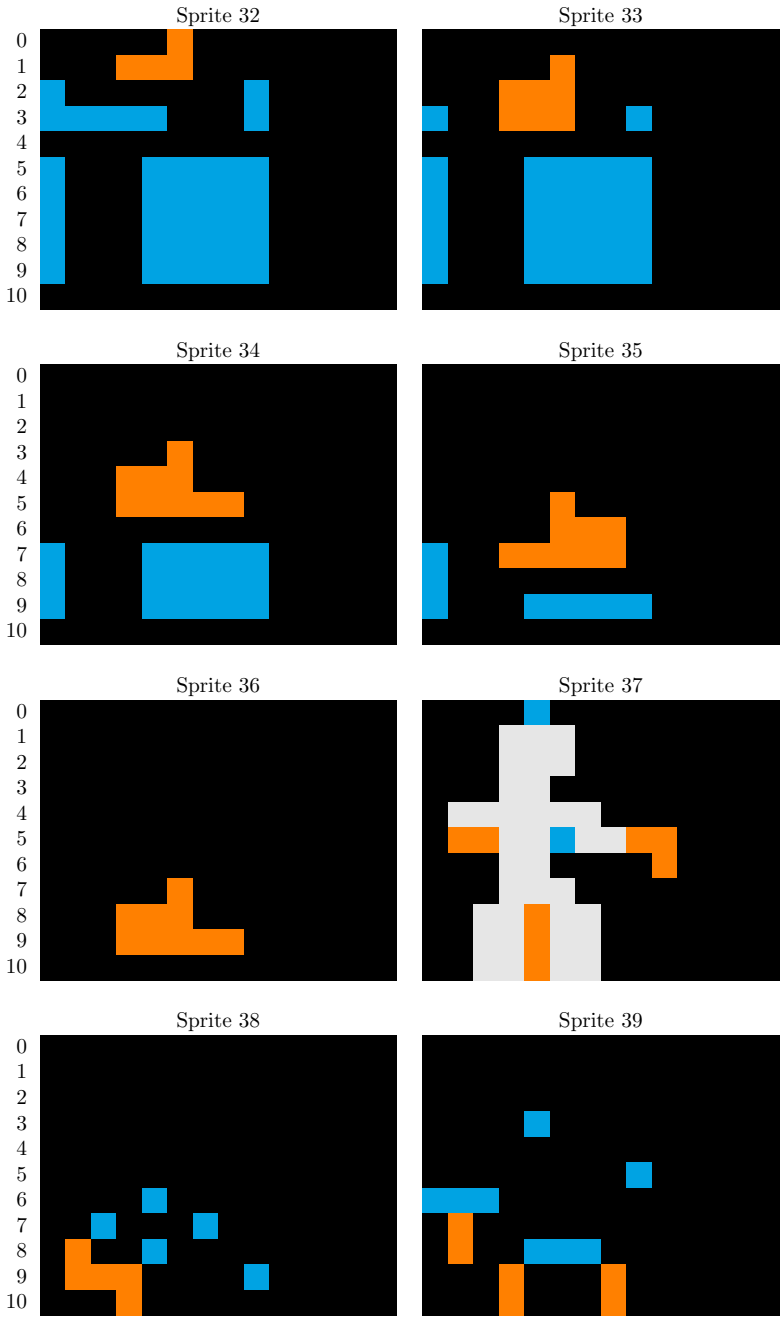
Defines:
SPRITE_DATA, used in chunk 26.

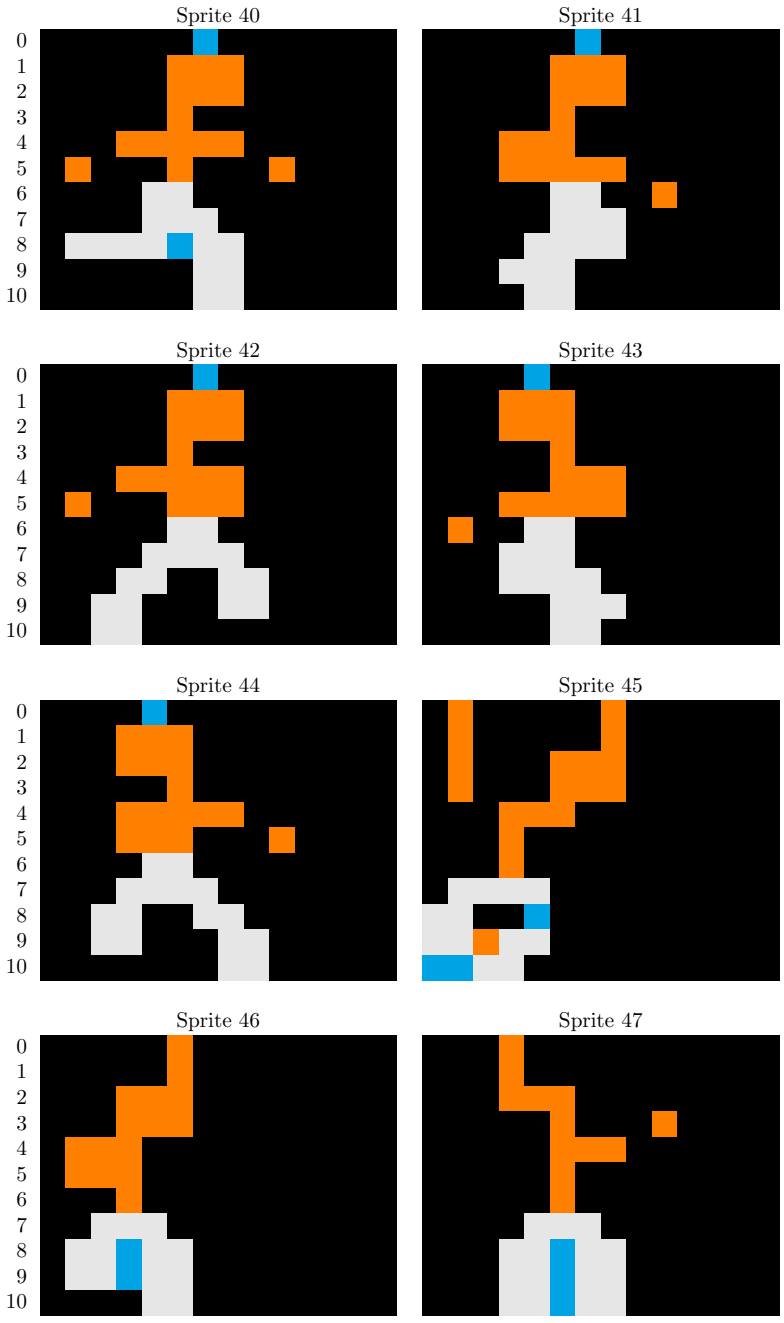

Sprite 0
Sprite 1
Sprite 2
Sprite 3
Sprite 4
Sprite 5
Sprite 6
Sprite 7
0
1
2
3
4
5
6
7
8
9
10

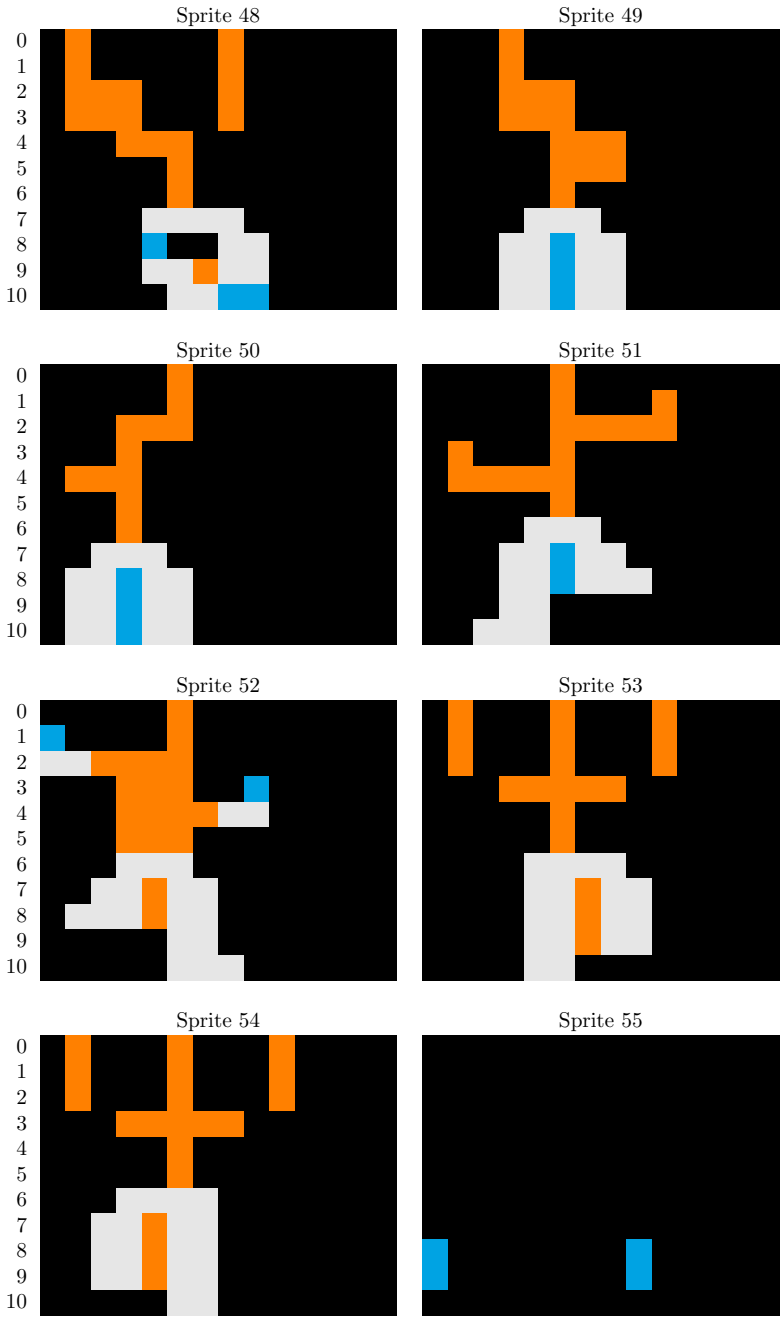

Sprite 8
Sprite 9
Sprite 10
Sprite 11
Sprite 12
Sprite 13
Sprite 14
Sprite 15
0
1
2
3
4
5
6
7
8
9
10

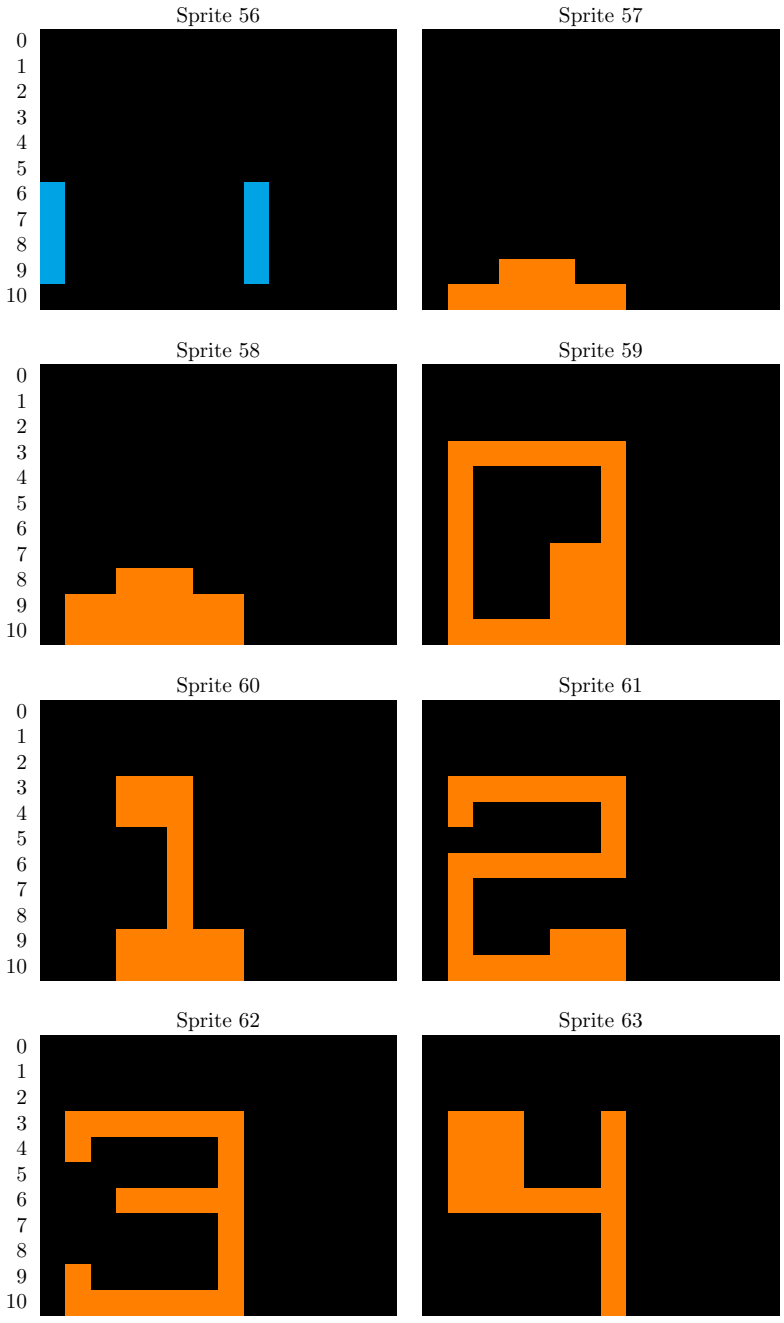

Sprite 16
Sprite 17
Sprite 18
Sprite 19
Sprite 20
Sprite 21
Sprite 22
Sprite 23
0
1
2
3
4
5
6
7
8
9
10


Sprite 24
Sprite 25
Sprite 26
Sprite 27
Sprite 28
Sprite 29
Sprite 30
Sprite 31
0
1
2
3
4
5
6
7
8
9
10

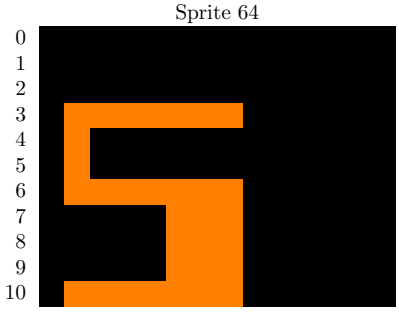

Sprite 32
Sprite 33
Sprite 34
Sprite 35
Sprite 36
Sprite 37
Sprite 38
Sprite 39
0
1
2
3
4
5
6
7
8
9
10


Sprite 40
Sprite 41
Sprite 42
Sprite 43
Sprite 44
Sprite 45
Sprite 46
Sprite 47
0
1
2
3
4
5
6
7
8
9
10

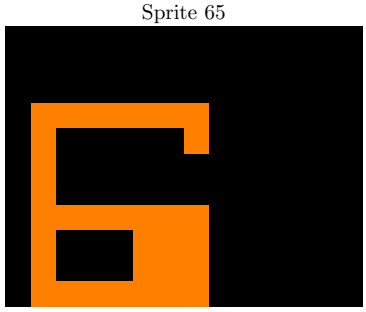

Sprite 48
Sprite 49
Sprite 50
Sprite 51
Sprite 52
Sprite 53
Sprite 54
Sprite 55
0
1
2
3
4
5
6
7
8
9
10


Sprite 56
Sprite 57
Sprite 58
Sprite 59
Sprite 60
Sprite 61
Sprite 62
Sprite 63
0
1
2
3
4
5
6
7
8
9
10

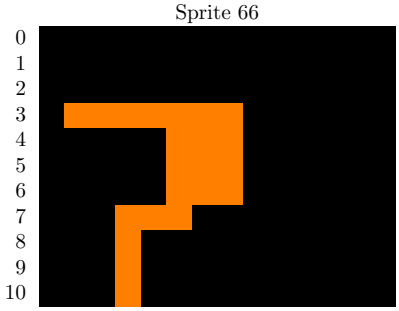
Sprite 64



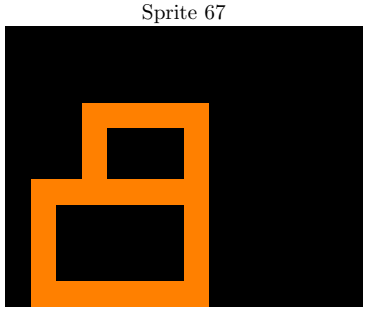
Sprite 65



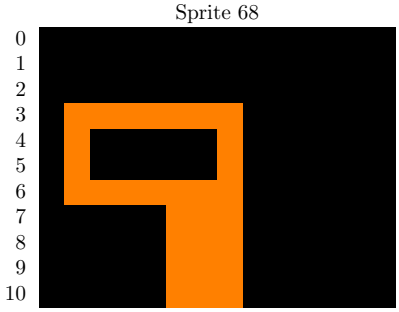
Sprite 66



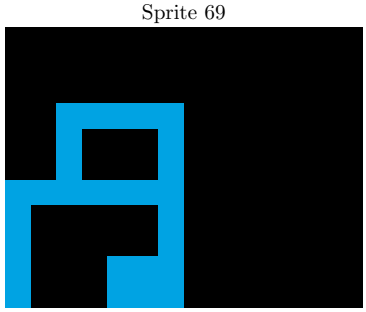
Sprite 67



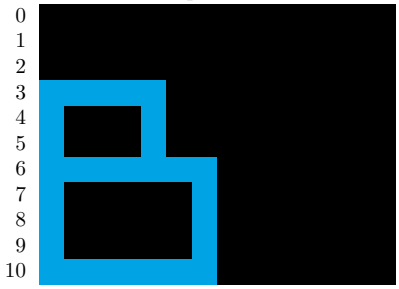
Sprite 68



Sprite 69

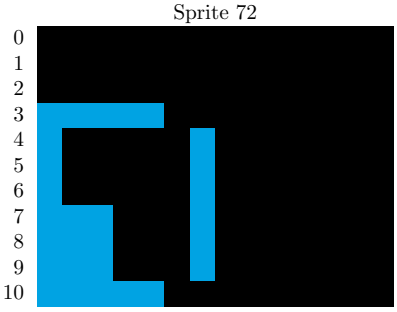


Sprite 70

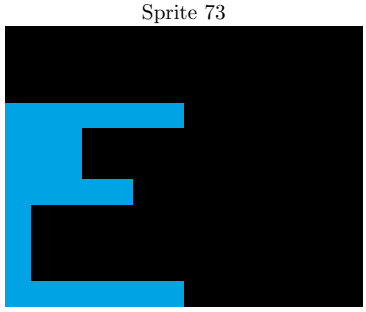


Sprite 71

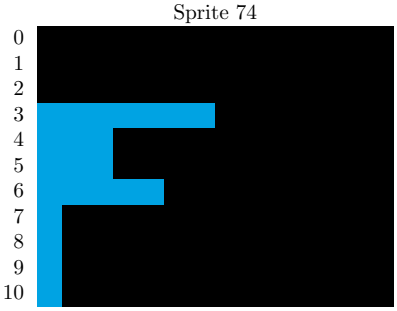
Sprite 72



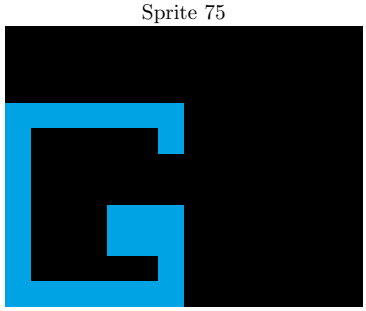
Sprite 73



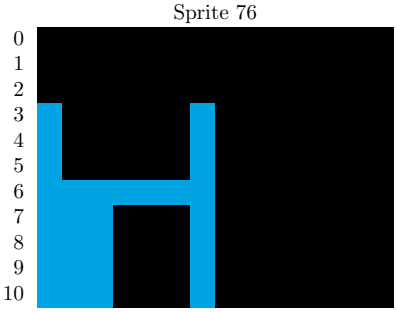
Sprite 74



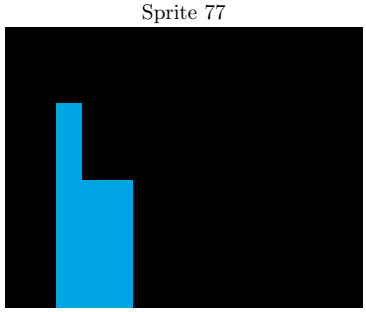
Sprite 75



Sprite 76



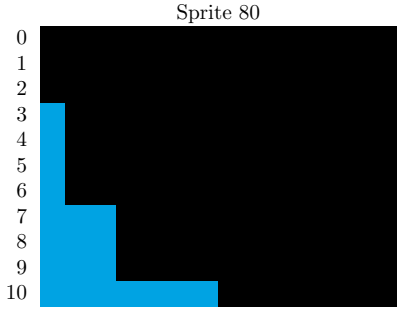
Sprite 77



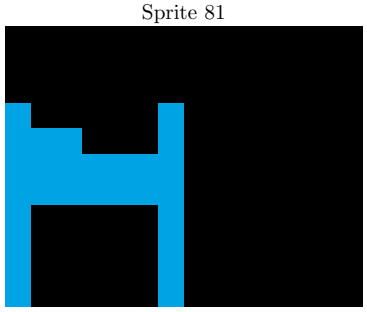
Sprite 78

Sprite 79

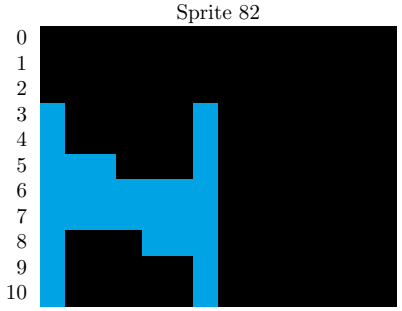
Sprite 80



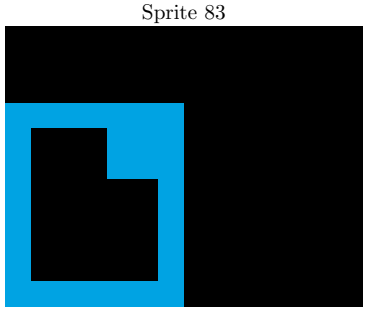
Sprite 81



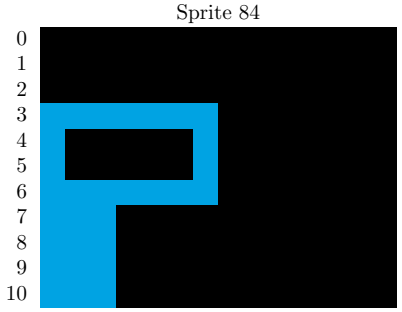
Sprite 82



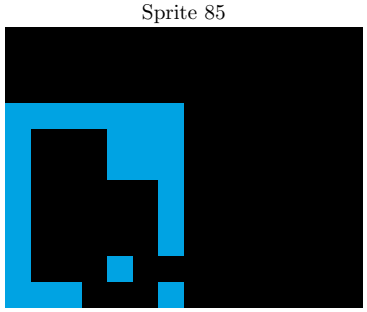
Sprite 83



Sprite 84



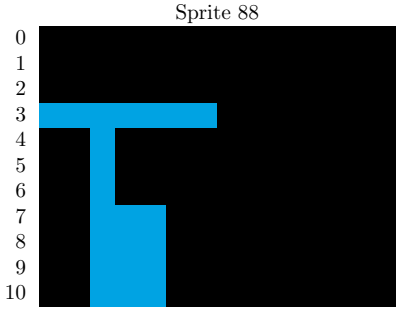
Sprite 85



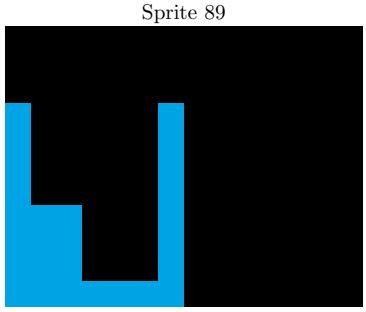
Sprite 86

Sprite 87

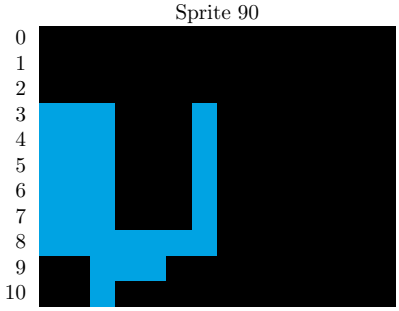
Sprite 88



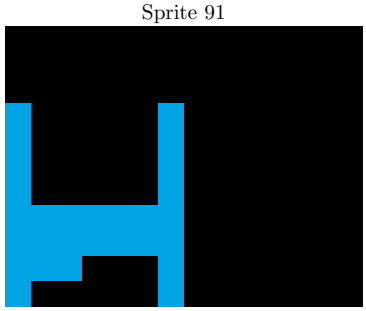
Sprite 89



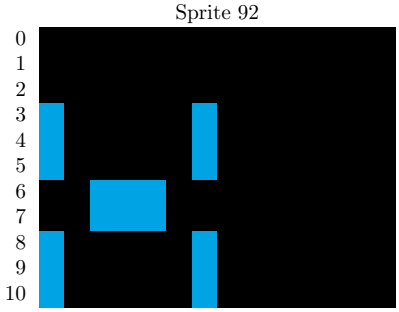
Sprite 90



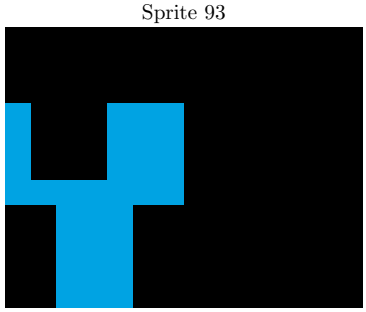
Sprite 91



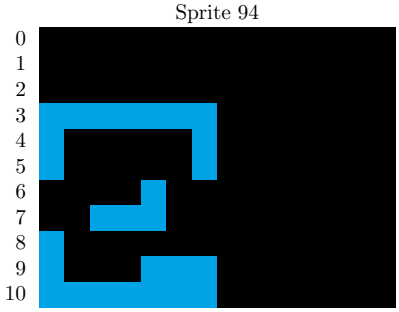
Sprite 92



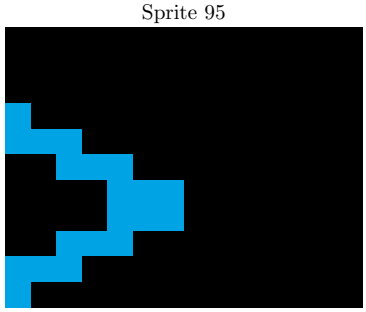
Sprite 93

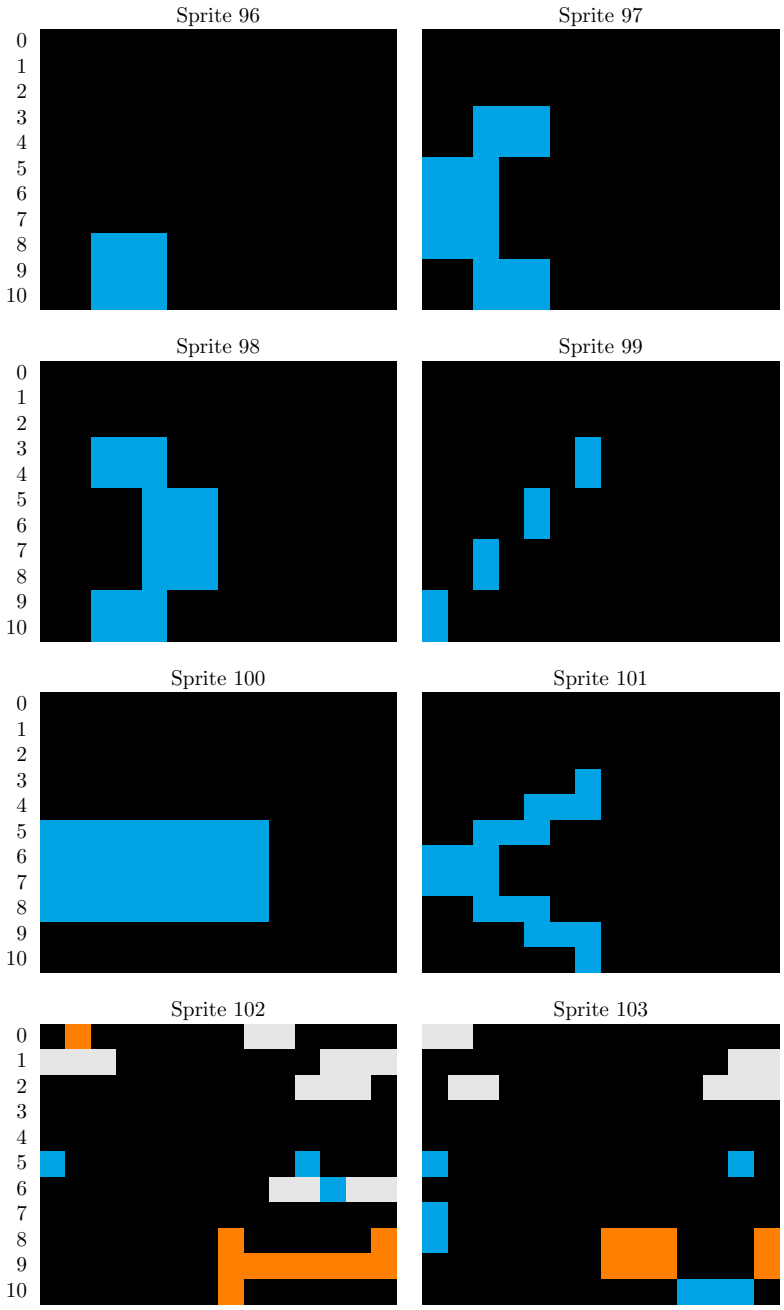


Sprite 94



Sprite 95





22 ⟨*defines* 4⟩+≡ (281) ◁4 23▷

```
SPRITE_EMPTY        EQU     #$00
SPRITE_BRICK        EQU     #$01
SPRITE_STONE        EQU     #$02
SPRITE_LADDER       EQU     #$03
```

```
SPRITE_ROPE         EQU     #$04
SPRITE_TRAP         EQU     #$05
SPRITE_INVISIBLE_LADDER       EQU     #$06
SPRITE_GOLD         EQU     #$07
SPRITE_GUARD        EQU     #$08
SPRITE_PLAYER       EQU     #$09
SPRITE_ALLWHITE     EQU     #$0A
SPRITE_BRICK_FILL0  EQU     #$37
SPRITE_BRICK_FILL1  EQU     #$38
SPRITE_GUARD_EGG0   EQU     #$39
SPRITE_GUARD_EGG1   EQU     #$3A
```

3.3 Shifting sprites

This is all very good if we're going to draw sprites exactly on 7-pixel boundaries, but what if we want to draw them starting at other columns? In general, such a shifted sprite would straddle three bytes, and Lode Runner sets aside an area of memory at the end of zero page for 11 rows of three bytes that we'll write to when we want to compute the data for a shifted sprite.

23 ⟨*defines* 4⟩+≡ (281) ◁22 25c▷

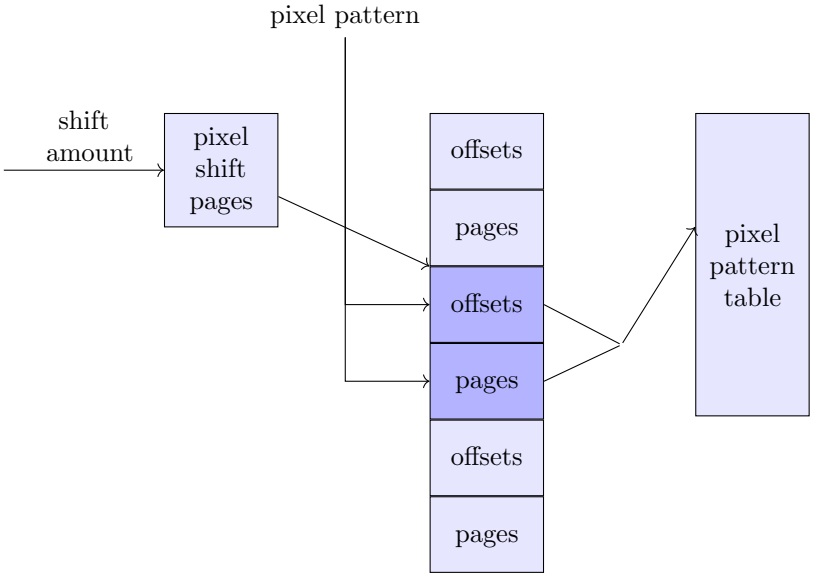
```
BLOCK_DATA      EQU     $DF     ; 33 bytes
```

Defines:
BLOCK_DATA, used in chunks 26, 35, 38, and 41.

Lode Runner also contains tables which show how to shift any arbitrary 7-pixel pattern right by any amount from zero to six pixels.

For example, suppose we start with a pixel pattern of `0110001`, and we want to shift that right by three bits. The 14-bit result would be `0000110 0010000`. However, we have to break that up into bytes, reverse the bits (remember that each byte's bits are output as pixels least significant bit first), and set their high bits, so we end up with `10110000 10000100`.

Now, given a shift amount and a pixel pattern, we should be able to find the two-byte shifted pattern. Lode Runner accomplishes this with table lookups as follows:



The pixel pattern table is a table of every possible pattern of 7 consecutive pixels spread out over two bytes. This table is 512 entries, each entry being two bytes. A naive table would have redundancy. For example the pattern `0000100` starting at column 0 is exactly the same as the pattern `0001000` starting at column 1. This table eliminates that redundancy.

24 ⟨*tables* 9⟩+≡ (281) ◁9 25a▷

```
    ORG     $A900
PIXEL_PATTERN_TABLE:
    INCLUDE "pixel_pattern_table.asm"
```

Defines:
`PIXEL_PATTERN_TABLE`, never used.

Now we just need tables which index into PIXEL_PATTERN_TABLE for every 7-pixel pattern and shift value. This table works by having the page number for the shifted pixel pattern at index `shift * 0x100 + 0x80 + pattern` and the offset at index `shift * 0x100 + pattern`.

25a ⟨*tables* 9⟩+≡ (281) ◁24 25b▷

```
    ORG     $A200
PIXEL_SHIFT_TABLE:
    INCLUDE "pixel_shift_table.asm"
```

Defines:
PIXEL_SHIFT_TABLE, never used.

Rather than multiplying the shift value by 0x100, we instead define another table which holds the page numbers for the shift tables for each shift value.

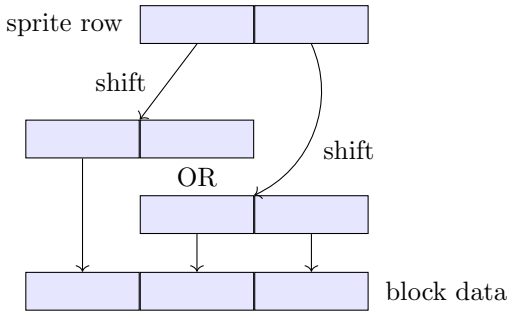
25b ⟨*tables* 9⟩+≡ (281) ◁25a 28a▷

```
    ORG     $84C1
PIXEL_SHIFT_PAGES:
    HEX     A2 A3 A4 A5 A6 A7 A8
```

Defines:
PIXEL_SHIFT_PAGES, used in chunk 26.

So we can get shifted pixels by indexing into all these tables.

Now we can define a routine that will take a sprite number and a pixel shift amount, and write the shifted pixel data into the BLOCK_DATA area. The routine first shifts the first byte of the sprite into a two-byte area. Then it shifts the second byte of the sprite, and combines that two-byte result with the first. Thus, we shift two bytes of sprite data into a three-byte result.



Rather than load addresses from the tables and store them, the routine modifies its own instructions with those addresses.

25c ⟨*defines* 4⟩+≡ (281) ◁23 28b▷

```
ROW_COUNT       EQU     $1D
SPRITE_NUM      EQU     $1E
```

Defines:
ROW_COUNT, used in chunks 26, 35, 38, 41, and 233.
SPRITE_NUM, used in chunks 26, 35, 38, 41, 116, 135b, 139, 187b, and 265.

26 ⟨*routines* 5⟩+≡ (281) ◁5 28c▷

```
      ORG     $8438
  COMPUTE_SHIFTED_SPRITE:
      SUBROUTINE
      ; Enter routine with X set to pixel shift amount and
      ; SPRITE_NUM containing the sprite number to read.

  .offset_table       EQU $A000               ; Target addresses in read
  .page_table         EQU $A080               ; instructions. The only truly
  .shift_ptr_byte0    EQU $A000               ; necessary value here is the
  .shift_ptr_byte1    EQU $A000               ; 0x80 in .shift_ptr_byte0.

      LDA     #$0B                            ; 11 rows
      STA     ROW_COUNT
      LDA     #<SPRITE_DATA
      STA     TMP_PTR
      LDA     #>SPRITE_DATA
      STA     TMP_PTR+1                       ; TMP_PTR = SPRITE_DATA
      LDA     PIXEL_SHIFT_PAGES,X
      STA     .rd_offset_table + 2
      STA     .rd_page_table + 2
      STA     .rd_offset_table2 + 2
      STA     .rd_page_table2 + 2             ; Fix up pages in lookup instructions
                                              ; based on shift amount (X).

      LDX     #$00                            ; X is the offset into BLOCK_DATA.

  .loop:                                      ; === LOOP === (over all 11 rows)
      LDY     SPRITE_NUM
      LDA     (TMP_PTR),Y
      TAY                                     ; Get sprite pixel data.

  .rd_offset_table:
      LDA     .offset_table,Y                 ; Load offset for shift amount.
      STA     .rd_shift_ptr_byte0 + 1
      CLC
      ADC     #$01
      STA     .rd_shift_ptr_byte1 + 1         ; Fix up instruction offsets with it.
  .rd_page_table:
      LDA     .page_table,Y                   ; Load page for shift amount.
      STA     .rd_shift_ptr_byte0 + 2
      STA     .rd_shift_ptr_byte1 + 2         ; Fix up instruction page with it.

  .rd_shift_ptr_byte0:
      LDA     .shift_ptr_byte0                ; Read shifted pixel data byte 0
      STA     BLOCK_DATA,X                    ; and store in block data byte 0.
  .rd_shift_ptr_byte1:
      LDA     .shift_ptr_byte1                ; Read shifted pixel data byte 1
      STA     BLOCK_DATA+1,X                  ; and store in block data byte 1.
```

```
    LDA     TMP_PTR
    CLC
    ADC     #$68
    STA     TMP_PTR
    LDA     TMP_PTR+1
    ADC     #$00
    STA     TMP_PTR+1                       ; TMP_PTR++

    ; Now basically do the same thing with the second sprite byte

    LDY     SPRITE_NUM
    LDA     (TMP_PTR),Y
    TAY                                     ; Get sprite pixel data.

.rd_offset_table2:
    LDA     .offset_table,Y                 ; Load offset for shift amount.
    STA     .rd_shift_ptr2_byte0 + 1
    CLC
    ADC     #$01
    STA     .rd_shift_ptr2_byte1 + 1        ; Fix up instruction offsets with it.
.rd_page_table2:
    LDA     .page_table,Y                   ; Load page for shift amount.
    STA     .rd_shift_ptr2_byte0 + 2
    STA     .rd_shift_ptr2_byte1 + 2        ; Fix up instruction page with it.

.rd_shift_ptr2_byte0:
    LDA     .shift_ptr_byte0                ; Read shifted pixel data byte 0
    ORA     BLOCK_DATA+1,X                  ; OR with previous block data byte 1
    STA     BLOCK_DATA+1,X                  ; and store in block data byte 1.
.rd_shift_ptr2_byte1:
    LDA     .shift_ptr_byte1                ; Read shifted pixel data byte 1
    STA     BLOCK_DATA+2,X                  ; and store in block data byte 2.

    LDA     TMP_PTR
    CLC
    ADC     #$68
    STA     TMP_PTR
    LDA     TMP_PTR+1
    ADC     #$00
    STA     TMP_PTR+1                       ; TMP_PTR++

    INX
    INX
    INX                                     ; X += 3
    DEC     ROW_COUNT                       ; ROW_COUNT--
    BNE     .loop                           ; loop while ROW_COUNT > 0
    RTS
```

Defines:
COMPUTE_SHIFTED_SPRITE, used in chunks 35, 38, and 41.
Uses BLOCK_DATA 23, PIXEL_SHIFT_PAGES 25b, ROW_COUNT 25c, SPRITE_DATA 9, SPRITE_NUM 25c,

and TMP_PTR 4.

3.4 Memory mapped graphics

Within a screen row, consecutive bytes map to consecutive pixels. However, rows themselves are not consecutive in memory.

To make it easy to convert a row number from 0 to 191 to a base address, Lode Runner has a table and a routine to use that table.

28a ⟨*tables* 9⟩+≡ (281) ◁25b 30▷

```
    ORG     $1A85
ROW_TO_OFFSET_LO:
    INCLUDE "row_to_offset_lo_table.asm"
ROW_TO_OFFSET_HI:
    INCLUDE "row_to_offset_hi_table.asm"
```

Defines:
ROW_TO_OFFSET_HI, used in chunks 28c and 29a.
ROW_TO_OFFSET_LO, used in chunks 28c and 29a.

28b ⟨*defines* 4⟩+≡ (281) ◁25c 29b▷

```
ROW_ADDR        EQU     $0C     ; 2 bytes
ROW_ADDR2       EQU     $0E     ; 2 bytes
HGR_PAGE        EQU     $1F     ; 0x20 for HGR1, 0x40 for HGR2
```

Defines:
HGR_PAGE, used in chunks 28c, 35, 124, and 254.
ROW_ADDR, used in chunks 28c, 29a, 35, 38, 41, 86, 98, 109, 125, and 256.
ROW_ADDR2, used in chunks 29a, 38, 41, 86, and 98.

28c ⟨*routines* 5⟩+≡ (281) ◁26 29a▷

```
    ORG     $7A31
ROW_TO_ADDR:
    SUBROUTINE
    ; Enter routine with Y set to row. Base address
    ; (for column 0) will be placed in ROW_ADDR.

    LDA     ROW_TO_OFFSET_LO,Y
    STA     ROW_ADDR
    LDA     ROW_TO_OFFSET_HI,Y
    ORA     HGR_PAGE
    STA     ROW_ADDR+1
    RTS
```

Defines:
ROW_TO_ADDR, used in chunks 35, 125, and 256.
Uses HGR_PAGE 28b, ROW_ADDR 28b, ROW_TO_OFFSET_HI 28a, and ROW_TO_OFFSET_LO 28a.

There's also a routine to load the address for both page 1 and page 2.

29a ⟨*routines* 5⟩+≡ (281) ◁28c 31a▷

```
      ORG     $7A3E
  ROW_TO_ADDR_FOR_BOTH_PAGES:
      SUBROUTINE
      ; Enter routine with Y set to row. Base address
      ; (for column 0) will be placed in ROW_ADDR (for page 1)
      ; and ROW_ADDR2 (for page 2).

      LDA     ROW_TO_OFFSET_LO,Y
      STA     ROW_ADDR
      STA     ROW_ADDR2
      LDA     ROW_TO_OFFSET_HI,Y
      ORA     #$20
      STA     ROW_ADDR+1
      EOR     #$60
      STA     ROW_ADDR2+1
      RTS
```

Defines:
ROW_TO_ADDR_FOR_BOTH_PAGES, used in chunks 38, 41, and 94–97.
Uses ROW_ADDR 28b, ROW_ADDR2 28b, ROW_TO_OFFSET_HI 28a, and ROW_TO_OFFSET_LO 28a.

Lode Runner's screens are organized into 28 sprites across by 17 sprites down. To convert between sprite coordinates and screen coordinates and vice-versa, we use tables and lookup routines. Each sprite is 10 pixels across by 11 pixels down.

Note that the last row is used for the status, so actually the game screen is 16 sprites vertically.

29b ⟨*defines* 4⟩+≡ (281) ◁28b 34a▷

```
  MAX_GAME_COL        EQU     #27     ; 0x1B
  MAX_GAME_ROW        EQU     #15     ; 0x0F
```

30 ⟨*tables* 9⟩+≡ (281) ◁28a 32b▷

```
    ORG     $1C35
HALF_SCREEN_COL_TABLE:
    ; 28 cols of 5 double-pixels each
    HEX     00 05 0a 0f 14 19 1e 23 28 2d 32 37 3c 41 46 4b
    HEX     50 55 5a 5f 64 69 6e 73 78 7d 82 87
SCREEN_ROW_TABLE:
    ; 17 rows of 11 pixels each
    HEX     00 0B 16 21 2C 37 42 4D 58 63 6E 79 84 8F 9A A5
    HEX     B5
COL_BYTE_TABLE:
    ; Byte number
    HEX     00 01 02 04 05 07 08 0A 0B 0C 0E 0F 11 12 14 15
    HEX     16 18 19 1B 1C 1E 1F 20 22 23 25 26
COL_SHIFT_TABLE:
    ; Right shift amount
    HEX     00 03 06 02 05 01 04 00 03 06 02 05 01 04 00 03
    HEX     06 02 05 01 04 00 03 06 02 05 01 04
HALF_SCREEN_COL_BYTE_TABLE:
    HEX     00 00 00 00 01 01 01 02 02 02 02 03 03 03 04 04
    HEX     04 04 05 05 05 06 06 06 06 07 07 07 08 08 08 08
    HEX     09 09 09 0A 0A 0A 0A 0B 0B 0B 0C 0C 0C 0C 0D 0D
    HEX     0D 0E 0E 0E 0E 0F 0F 0F 10 10 10 10 11 11 11 12
    HEX     12 12 12 13 13 13 14 14 14 14 15 15 15 16 16 16
    HEX     16 17 17 17 18 18 18 18 19 19 19 1A 1A 1A 1A 1B
    HEX     1B 1B 1C 1C 1C 1C 1D 1D 1D 1E 1E 1E 1E 1F 1F 1F
    HEX     20 20 20 20 21 21 21 22 22 22 22 23 23 23 24 24
    HEX     24 24 25 25 25 26 26 26 26 27 27 27
HALF_SCREEN_COL_SHIFT_TABLE:
    HEX     00 02 04 06 01 03 05 00 02 04 06 01 03 05 00 02
    HEX     04 06 01 03 05 00 02 04 06 01 03 05 00 02 04 06
    HEX     01 03 05 00 02 04 06 01 03 05 00 02 04 06 01 03
    HEX     05 00 02 04 06 01 03 05 00 02 04 06 01 03 05 00
    HEX     02 04 06 01 03 05 00 02 04 06 01 03 05 00 02 04
    HEX     06 01 03 05 00 02 04 06 01 03 05 00 02 04 06 01
    HEX     03 05 00 02 04 06 01 03 05 00 02 04 06 01 03 05
    HEX     00 02 04 06 01 03 05 00 02 04 06 01 03 05 00 02
    HEX     04 06 01 03 05 00 02 04 06 01 03 05
```

Defines:
COL_BYTE_TABLE, used in chunks 31b and 35.
COL_SHIFT_TABLE, used in chunks 31b and 35.
HALF_SCREEN_COL_BYTE_TABLE, used in chunk 32a.
HALF_SCREEN_COL_SHIFT_TABLE, used in chunk 32a.
HALF_SCREEN_COL_TABLE, used in chunk 31a.
SCREEN_ROW_TABLE, used in chunks 31a and 35.

Here is the routine to return the screen coordinates for the given sprite coordinates. The reason that GET_SCREEN_COORDS_FOR returns half the screen column coordinate is that otherwise the screen column coordinate wouldn't fit in a register.

31a ⟨*routines* 5⟩+≡ (281) ◁29a 31b▷

```
      ORG     $885D
  GET_SCREEN_COORDS_FOR:
      SUBROUTINE
      ; Enter routine with Y set to sprite row (0-16) and
      ; X set to sprite column (0-27). On return, Y will be set to
      ; screen row, and X is set to half screen column.

      LDA     SCREEN_ROW_TABLE,Y
      PHA
      LDA     HALF_SCREEN_COL_TABLE,X
      TAX                         ; X = HALF_SCREEN_COL_TABLE[X]
      PLA
      TAY                         ; Y = SCREEN_ROW_TABLE[Y]
      RTS
```

Defines:
GET_SCREEN_COORDS_FOR, used in chunks 33, 35, 126, 137, 168, 171, 179, 184, 193, and 212.
Uses HALF_SCREEN_COL_TABLE 30 and SCREEN_ROW_TABLE 30.

This routine takes a sprite column and converts it to the memory-mapped byte offset and right-shift amount.

31b ⟨*routines* 5⟩+≡ (281) ◁31a 32a▷

```
      ORG     $8868
  GET_BYTE_AND_SHIFT_FOR_COL:
      SUBROUTINE
      ; Enter routine with X set to sprite column. On
      ; return, A will be set to screen column byte number
      ; and X will be set to an additional right shift amount.

      LDA     COL_BYTE_TABLE,X
      PHA                         ; A = COL_BYTE_TABLE[X]
      LDA     COL_SHIFT_TABLE,X
      TAX                         ; X = COL_SHIFT_TABLE[X]
      PLA
      RTS
```

Defines:
GET_BYTE_AND_SHIFT_FOR_COL, used in chunk 35.
Uses COL_BYTE_TABLE 30 and COL_SHIFT_TABLE 30.

This routine takes half the screen column coordinate and converts it to the memory-mapped byte offset and right-shift amount.

32a ⟨*routines* 5⟩+≡ (281) ◁31b 33a▷

```
    ORG     $8872
GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL:
    SUBROUTINE
    ; Enter routine with X set to half screen column. On
    ; return, A will be set to screen column byte number
    ; and X will be set to an additional right shift amount.

    LDA     HALF_SCREEN_COL_BYTE_TABLE,X
    PHA                         ; A = HALF_SCREEN_COL_BYTE_TABLE[X]
    LDA     HALF_SCREEN_COL_SHIFT_TABLE,X
    TAX                         ; X = HALF_SCREEN_COL_SHIFT_TABLE[X]
    PLA
    RTS
```

Defines:
GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL, used in chunks 38 and 41.
Uses HALF_SCREEN_COL_BYTE_TABLE 30 and HALF_SCREEN_COL_SHIFT_TABLE 30.

We also have some utility routines that let us take a sprite row or column and get its screen row or half column, but offset in either row or column by anywhere from -2 to +2.

32b ⟨*tables* 9⟩+≡ (281) ◁30 33b▷

```
    ORG     $888A
ROW_OFFSET_TABLE:
    HEX     FB FD 00 02 04
```

Defines:
ROW_OFFSET_TABLE, used in chunk 33a.

33a ⟨*routines* 5⟩+≡ (281) ◁32a 33c▷

```
      ORG     $887C
  GET_SCREEN_ROW_OFFSET_IN_X_FOR:
      SUBROUTINE
      ; Enter routine with X set to offset+2 (in double-pixels) and
      ; Y set to sprite row. On return, X will retain its value and
      ; Y will be set to the screen row.

      TXA
      PHA
      JSR     GET_SCREEN_COORDS_FOR
      PLA
      TAX                                ; Restore X
      TYA
      CLC
      ADC     ROW_OFFSET_TABLE,X
      TAY
      RTS
```

Defines:
  GET_SCREEN_ROW_OFFSET_IN_X_FOR, used in chunks 135b and 187b.
Uses GET_SCREEN_COORDS_FOR 31a and ROW_OFFSET_TABLE 32b.

33b ⟨*tables* 9⟩+≡ (281) ◁32b 34b▷

```
      ORG     $889D
  COL_OFFSET_TABLE:
      HEX     FE FF 00 01 02
```

Defines:
  COL_OFFSET_TABLE, used in chunk 33c.

33c ⟨*routines* 5⟩+≡ (281) ◁33a 35▷

```
      ORG     $888F
  GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR:
      SUBROUTINE
      ; Enter routine with Y set to offset+2 (in double-pixels) and
      ; X set to sprite column. On return, Y will retain its value and
      ; X will be set to the half screen column.

      TYA
      PHA
      JSR     GET_SCREEN_COORDS_FOR
      PLA
      TAY                                ; Restore Y
      TXA
      CLC
      ADC     COL_OFFSET_TABLE,Y
      TAX
      RTS
```

Defines:
  GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR, used in chunks 135b and 187b.
Uses COL_OFFSET_TABLE 33b and GET_SCREEN_COORDS_FOR 31a.

Now we can finally write the routines that draw a sprite on the screen. We have one routine that draws a sprite at a given game row and game column. There are two entry points, one to draw on HGR1, and one for HGR2.

34a ⟨*defines* 4⟩+≡ (281) ◁29b 40▷

```
ROWNUM          EQU     $1B
COLNUM          EQU     $1C
MASK0           EQU     $50
MASK1           EQU     $51
COL_SHIFT_AMT   EQU     $71
GAME_COLNUM     EQU     $85
GAME_ROWNUM     EQU     $86
```

Defines:
COL_SHIFT_AMT, used in chunks 35, 38, and 41.
COLNUM, used in chunks 35, 38, and 41.
GAME_COLNUM, used in chunks 35, 46a, 48a, 51, 53, 74, 81b, 85b, 87, 114, 116, 119b, 126, 137, 145, 168, 171, 174, 179, 184, 188, 193, 212, 233, 237, 250, 259, 265, 266, and 268.
GAME_ROWNUM, used in chunks 35, 46a, 51, 53, 77, 82–85, 87, 112b, 113, 115a, 116, 119b, 124–26, 131a, 132a, 134d, 137, 145, 168, 171, 174, 179, 184, 188, 193, 212, 233, 237, 250, 256, 259, 265, 266, and 268.
MASK0, used in chunks 35 and 231.
MASK1, used in chunk 35.
ROWNUM, used in chunks 35, 38, 41, and 201.

34b ⟨*tables* 9⟩+≡ (281) ◁33b 60b▷

```
        ORG     $8328
PIXEL_MASK0:
        BYTE    %00000000
        BYTE    %00000001
        BYTE    %00000011
        BYTE    %00000111
        BYTE    %00001111
        BYTE    %00011111
        BYTE    %00111111
PIXEL_MASK1:
        BYTE    %11111000
        BYTE    %11110000
        BYTE    %11100000
        BYTE    %11000000
        BYTE    %10000000
        BYTE    %11111110
        BYTE    %11111100
```

Defines:
PIXEL_MASK0, used in chunk 35.
PIXEL_MASK1, used in chunk 35.

35 ⟨*routines 5*⟩+≡ (281) ◁33c 90▷

```
        ORG     $82AA
    DRAW_SPRITE_PAGE1:
        SUBROUTINE
        ; Enter routine with A set to sprite number to draw,
        ; GAME_ROWNUM set to the row to draw it at, and GAME_COLNUM
        ; set to the column to draw it at.

        STA     SPRITE_NUM
        LDA     #$20                ; Page number for HGR1
        BNE     DRAW_SPRITE         ; Actually unconditional jump

    DRAW_SPRITE_PAGE2:
        SUBROUTINE
        ; Enter routine with A set to sprite number to draw,
        ; GAME_ROWNUM set to the row to draw it at, and GAME_COLNUM
        ; set to the column to draw it at.

        STA     SPRITE_NUM
        LDA     #$40                ; Page number for HGR2
        ; fallthrough

    DRAW_SPRITE:
        STA     HGR_PAGE
        LDY     GAME_ROWNUM
        JSR     GET_SCREEN_COORDS_FOR
        STY     ROWNUM              ; ROWNUM = SCREEN_ROW_TABLE[GAME_ROWNUM]

        LDX     GAME_COLNUM
        JSR     GET_BYTE_AND_SHIFT_FOR_COL
        STA     COLNUM              ; COLNUM = COL_BYTE_TABLE[GAME_COLNUM]
        STX     COL_SHIFT_AMT       ; COL_SHIFT_AMT = COL_SHIFT_TABLE[GAME_COLNUM]

        LDA     PIXEL_MASK0,X
        STA     MASK0               ; MASK0 = PIXEL_MASK0[COL_SHIFT_AMT]
        LDA     PIXEL_MASK1,X
        STA     MASK1               ; MASK1 = PIXEL_MASK1[COL_SHIFT_AMT]

        JSR     COMPUTE_SHIFTED_SPRITE

        LDA     #$0B
        STA     ROW_COUNT
        LDX     #$00
        LDA     COL_SHIFT_AMT
        CMP     #$05
        BCS     .need_3_bytes       ; If COL_SHIFT_AMT >= 5, we need to alter three screen bytes,
                                    ; otherwise just two bytes.

    .loop1:
        LDY     ROWNUM
```

```
    JSR     ROW_TO_ADDR
    LDY     COLNUM
    LDA     (ROW_ADDR),Y
    AND     MASK0
    ORA     BLOCK_DATA,X
    STA     (ROW_ADDR),Y        ; screen[COLNUM] = screen[COLNUM] & MASK0 | BLOCK_DATA[i]

    INX                         ; X++
    INY                         ; Y++
    LDA     (ROW_ADDR),Y
    AND     MASK1
    ORA     BLOCK_DATA,X
    STA     (ROW_ADDR),Y        ; screen[COLNUM+1] = screen[COLNUM+1] & MASK1 | BLOCK_DATA[i+1]

    INX
    INX                         ; X += 2
    INC     ROWNUM              ; ROWNUM++
    DEC     ROW_COUNT           ; ROW_COUNT--
    BNE     .loop1              ; loop while ROW_COUNT > 0
    RTS

.need_3_bytes
    LDY     ROWNUM
    JSR     ROW_TO_ADDR
    LDY     COLNUM
    LDA     (ROW_ADDR),Y
    AND     MASK0
    ORA     BLOCK_DATA,X
    STA     (ROW_ADDR),Y        ; screen[COLNUM] = screen[COLNUM] & MASK0 | BLOCK_DATA[i]

    INX                         ; X++
    INY                         ; Y++
    LDA     BLOCK_DATA,X
    STA     (ROW_ADDR),Y        ; screen[COLNUM+1] = BLOCK_DATA[i+1]

    INX                         ; X++
    INY                         ; Y++
    LDA     (ROW_ADDR),Y
    AND     MASK1
    ORA     BLOCK_DATA,X
    STA     (ROW_ADDR),Y        ; screen[COLNUM+2] = screen[COLNUM+2] & MASK1 | BLOCK_DATA[i+2]

    INX                         ; X++
    INC     ROWNUM              ; ROWNUM++
    DEC     ROW_COUNT           ; ROW_COUNT--
    BNE     .need_3_bytes       ; loop while ROW_COUNT > 0
    RTS
```

Defines:
DRAW_SPRITE_PAGE1, used in chunks 46a, 48a, 71, 126, 168, 171, 174, 188, and 265.
DRAW_SPRITE_PAGE2, used in chunks 46a, 48a, 70, 85b, 87, 126, 137, 145, 174, 179, 184, 188,

193, and 212.
Uses BLOCK_DATA 23, COL_BYTE_TABLE 30, COL_SHIFT_AMT 34a, COL_SHIFT_TABLE 30, COLNUM 34a, COMPUTE_SHIFTED_SPRITE 26, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_BYTE_AND_SHIFT_FOR_COL 31b, GET_SCREEN_COORDS_FOR 31a, HGR_PAGE 28b, MASK0 34a, MASK1 34a, PIXEL_MASK0 34b, PIXEL_MASK1 34b, ROW_ADDR 28b, ROW_COUNT 25c, ROW_TO_ADDR 28c, ROWNUM 34a, SCREEN_ROW_TABLE 30, and SPRITE_NUM 25c.

There is a different routine which erases a sprite at a given screen coordinate. It does this by drawing the inverse of the sprite on page 1, then drawing the sprite data from page 2 (the background page) onto page 1.

Upon entry, the Y register needs to be set to the screen row coordinate (0-191). However, the X register needs to be set to half the screen column coordinate (0-139) because otherwise the maximum coordinate (279) wouldn't fit in a register.

38 ⟨*erase sprite at screen coordinate* 38⟩≡ (278)

```
    ORG     $8336
ERASE_SPRITE_AT_PIXEL_COORDS:
    SUBROUTINE
    ; Enter routine with A set to sprite number to draw,
    ; Y set to the screen row to erase it at, and X
    ; set to *half* the screen column to erase it at.

    STY     ROWNUM
    STA     SPRITE_NUM
    JSR     GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL
    STA     COLNUM
    STX     COL_SHIFT_AMT
    JSR     COMPUTE_SHIFTED_SPRITE

    LDX     #$0B
    STX     ROW_COUNT
    LDX     #$00
    LDA     COL_SHIFT_AMT
    CMP     #$05
    BCS     .need_3_bytes       ; If COL_SHIFT_AMT >= 5, we need to alter three screen bytes,
                                ; otherwise just two bytes.

.loop1:
    LDY     ROWNUM
    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES
    LDY     COLNUM
    LDA     BLOCK_DATA,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y            ; screen[COLNUM] =
                                    ;   (screen[COLNUM] & (BLOCK_DATA[i] ^ 0x7F)) | screen2[COLNUM]

    INX                             ; X++
    INY                             ; Y++
    LDA     BLOCK_DATA,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y            ; screen[COLNUM+1] =
                                    ;   (screen[COLNUM+1] & (BLOCK_DATA[i+1] ^ 0x7F)) | screen2[COLN
```

```
    INX                             ; X++
    INX                             ; X++
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .loop1
    RTS

.need_3_bytes:
    LDY     ROWNUM
    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES
    LDY     COLNUM
    LDA     BLOCK_DATA,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y            ; screen[COLNUM] =
                                    ;   (screen[COLNUM] & (BLOCK_DATA[i] ^ 0x7F)) | screen2[COLNUM]

    INX                             ; X++
    INY                             ; Y++
    LDA     BLOCK_DATA,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y            ; screen[COLNUM+1] =
                                    ;   (screen[COLNUM+1] & (BLOCK_DATA[i+1] ^ 0x7F)) | screen2[COLN

    INX                             ; X++
    INY                             ; Y++
    LDA     BLOCK_DATA,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y            ; screen[COLNUM+2] =
                                    ;   (screen[COLNUM+2] & (BLOCK_DATA[i+2] ^ 0x7F)) | screen2[COLN

    INX                             ; X++
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .need_3_bytes
    RTS
```

Defines:
ERASE_SPRITE_AT_PIXEL_COORDS, used in chunks 126, 137, 157, 159, 161, 164, 168, 171, 175, 184, 193, 204, 206, 208, and 210.
Uses BLOCK_DATA 23, COL_SHIFT_AMT 34a, COLNUM 34a, COMPUTE_SHIFTED_SPRITE 26, GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL 32a, ROW_ADDR 28b, ROW_ADDR2 28b, ROW_COUNT 25c, ROW_TO_ADDR_FOR_BOTH_PAGES 29a, ROWNUM 34a, and SPRITE_NUM 25c.

And then there's the corresponding routine to draw a sprite at the given coordinates. The routine also sets whether the active and the background screens differ in `SCREENS_DIFFER`.

40 ⟨*defines* 4⟩+≡ (281) ◁34a 45▷

```
SCREENS_DIFFER      EQU     $52
```

Defines:
`SCREENS_DIFFER`, used in chunks 41 and 43.

41 ⟨*draw sprite at screen coordinate* 41⟩≡ (278)

```
      ORG     $83A7
  DRAW_SPRITE_AT_PIXEL_COORDS:
      SUBROUTINE
      ; Enter routine with A set to sprite number to draw,
      ; Y set to the screen row to draw it at, and X
      ; set to *half* the screen column to draw it at.

      STY     ROWNUM
      STA     SPRITE_NUM
      JSR     GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL
      STA     COLNUM
      STX     COL_SHIFT_AMT
      JSR     COMPUTE_SHIFTED_SPRITE

      LDA     #$0B
      STA     ROW_COUNT
      LDX     #$00
      STX     SCREENS_DIFFER      ; SCREENS_DIFFER = 0
      LDA     COL_SHIFT_AMT
      CMP     #$05
      BCS     .need_3_bytes       ; If COL_SHIFT_AMT >= 5, we need to alter three screen bytes,
                                  ; otherwise just two bytes.

  .loop1:
      LDY     ROWNUM
      JSR     ROW_TO_ADDR_FOR_BOTH_PAGES
      LDY     COLNUM
      LDA     (ROW_ADDR),Y
      EOR     (ROW_ADDR2),Y
      AND     BLOCK_DATA,X
      ORA     SCREENS_DIFFER
      STA     SCREENS_DIFFER          ; SCREENS_DIFFER |=
                                      ;   ( (screen[COLNUM] ^ screen2[COLNUM]) & BLOCK_DATA[i])
      LDA     BLOCK_DATA,X
      ORA     (ROW_ADDR),Y
      STA     (ROW_ADDR),Y            ; screen[COLNUM] |= BLOCK_DATA[i]

      INX                             ; X++
      INY                             ; Y++
      LDA     (ROW_ADDR),Y
      EOR     (ROW_ADDR2),Y
      AND     BLOCK_DATA,X
      ORA     SCREENS_DIFFER
      STA     SCREENS_DIFFER          ; SCREENS_DIFFER |=
                                      ;   ( (screen[COLNUM+1] ^ screen2[COLNUM+1]) & BLOCK_DATA[i+1])
      LDA     BLOCK_DATA,X
      ORA     (ROW_ADDR),Y
      STA     (ROW_ADDR),Y            ; screen[COLNUM+1] |= BLOCK_DATA[i+1]
```

```
    INX                         ; X++
    INX                         ; X++
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .loop1
    RTS

.need_3_bytes:
    LDY     ROWNUM
    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES
    LDY     COLNUM
    LDA     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER      ; SCREENS_DIFFER |=
                                ;   ( (screen[COLNUM] ^ screen2[COLNUM]) & BLOCK_DATA[i])
    LDA     BLOCK_DATA,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM] |= BLOCK_DATA[i]

    INX                         ; X++
    INY                         ; Y++
    LDA     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER      ; SCREENS_DIFFER |=
                                ;   ( (screen[COLNUM+1] ^ screen2[COLNUM+1]) & BLOCK_DATA[i+1])
    LDA     BLOCK_DATA,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM+1] |= BLOCK_DATA[i+1]

    INX                         ; X++
    INY                         ; Y++
    LDA     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER      ; SCREENS_DIFFER |=
                                ;   ( (screen[COLNUM+2] ^ screen2[COLNUM+2]) & BLOCK_DATA[i+2])
    LDA     BLOCK_DATA,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM+2] |= BLOCK_DATA[i+2]

    INX                         ; X++
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .need_3_bytes
    RTS
```

Defines:
DRAW_SPRITE_AT_PIXEL_COORDS, used in chunks 43, 168, 171, 179, 188, 193, 204, 206, 208, and 212.
Uses BLOCK_DATA 23, COL_SHIFT_AMT 34a, COLNUM 34a, COMPUTE_SHIFTED_SPRITE 26, GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL 32a, ROW_ADDR 28b, ROW_ADDR2 28b, ROW_COUNT 25c, ROW_TO_ADDR_FOR_BOTH_PAGES 29a, ROWNUM 34a, SCREENS_DIFFER 40, and SPRITE_NUM 25c.

There is a special routine to draw the player sprite at the player's location. If the two pages at the player's location are different and the player didn't pick up gold (which would explain the difference), then the player is killed.

43 ⟨*draw player* 43⟩≡ (278)

```
      ORG     $6C02
  DRAW_PLAYER:
      SUBROUTINE

      JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
      JSR     DRAW_SPRITE_AT_PIXEL_COORDS
      LDA     SCREENS_DIFFER
      BEQ     .end
      LDA     DIDNT_PICK_UP_GOLD
      BEQ     .end
      LSR     ALIVE       ; Set player as dead
  .end
      RTS
```

Defines:
DRAW_PLAYER, used in chunks 157, 161, 164, 168, 171, and 175.
Uses ALIVE 111a, DIDNT_PICK_UP_GOLD 136b, DRAW_SPRITE_AT_PIXEL_COORDS 41, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, and SCREENS_DIFFER 40.

3.5 Printing strings

Now that we can put sprites onto the screen at any game coordinate, we can also have some routines that print strings. We saw above that we have letter and number sprites, plus some punctuation. Letters and punctuation are always blue, while numbers are always orange.

There is a basic routine to put a character at the current GAME_COLNUM and GAME_ROWNUM, incrementing this "cursor", and putting it at the beginning of the next line if we "print" a newline character.

We first define a routine to convert the ASCII code of a character to its sprite number. Lode Runner sets the high bit of the code to make it be treated as ASCII.

44 ⟨*char to sprite num* 44⟩≡ (278)

```
    ORG     $7B2A
CHAR_TO_SPRITE_NUM:
    SUBROUTINE
    ; Enter routine with A set to the ASCII code of the
    ; character to convert to sprite number, with the high bit set.
    ; The sprite number is returned in A.

    CMP     #$C1                    ; 'A' -> sprite 69
    BCC     .not_letter
    CMP     #$DB                    ; 'Z' -> sprite 94
    BCC     .letter

.not_letter:
    ; On return, we will subtract 0x7C from X to
    ; get the actual sprite. This is to make A-Z
    ; easier to handle.
    LDX     #$7C
    CMP     #$A0                    ; ' ' -> sprite 0
    BEQ     .end
    LDX     #$DB
    CMP     #$BE                    ; '>' -> sprite 95
    BEQ     .end
    INX
    CMP     #$AE                    ; '.' -> sprite 96
    BEQ     .end
    INX
    CMP     #$A8                    ; '(' -> sprite 97
    BEQ     .end
    INX
    CMP     #$A9                    ; ')' -> sprite 98
    BEQ     .end
    INX
    CMP     #$AF                    ; '/' -> sprite 99
    BEQ     .end
    INX
    CMP     #$AD                    ; '-' -> sprite 100
```

```
    BEQ     .end
    INX
    CMP     #$BC                    ; '<' -> sprite 101
    BEQ     .end
    LDA     #$10                    ; sprite 16: just one of the man sprites
    RTS

.end:
    TXA

.letter:
    SEC
    SBC     #$7C
    RTS
```

Defines:
CHAR_TO_SPRITE_NUM, used in chunks 46a and 233.

Now we can define the routine to put a character on the screen at the current position.

45 ⟨*defines* 4⟩+≡ (281) ◁40 46b▷

```
DRAW_PAGE   EQU     $87     ; 0x20 for page 1, 0x40 for page 2
```

Defines:
DRAW_PAGE, used in chunks 46a, 48a, 53, 119b, 123, 124, 233, 237, 259, 265, and 268.

46a ⟨*put char* 46a⟩≡ (278)

```
      ORG     $7B64
  PUT_CHAR:
      SUBROUTINE
      ; Enter routine with A set to the ASCII code of the
      ; character to put on the screen, with the high bit set.

      CMP     #$8D
      BEQ     NEWLINE                 ; If newline, do NEWLINE instead.
      JSR     CHAR_TO_SPRITE_NUM
      LDX     DRAW_PAGE
      CPX     #$40
      BEQ     .draw_to_page2

      JSR     DRAW_SPRITE_PAGE1
      INC     GAME_COLNUM
      RTS

  .draw_to_page2
      JSR     DRAW_SPRITE_PAGE2
      INC     GAME_COLNUM
      RTS

  NEWLINE:
      SUBROUTINE
      INC     GAME_ROWNUM
      LDA     #$00
      STA     GAME_COLNUM
      RTS
```

Defines:
NEWLINE, used in chunk 122b.
PUT_CHAR, used in chunks 47, 120c, 121b, and 233.
Uses CHAR_TO_SPRITE_NUM 44, DRAW_PAGE 45, DRAW_SPRITE_PAGE1 35, DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, and GAME_ROWNUM 34a.

The PUT_STRING routine uses PUT_CHAR to put a string on the screen. Rather than take an address pointing to a string, instead it uses the return address as the source for data. It then has to fix up the actual return address at the end to be just after the zero-terminating byte of the string.

46b ⟨*defines* 4⟩+≡ (281) ◁45 48b▷

```
  SAVED_RET_ADDR      EQU     $10     ; 2 bytes
```

Defines:
SAVED_RET_ADDR, used in chunks 47 and 58.

47 ⟨*put string* 47⟩≡ (278)

```
        ORG     $86E0
    PUT_STRING:
        SUBROUTINE

        PLA
        STA     SAVED_RET_ADDR
        PLA
        STA     SAVED_RET_ADDR+1
        BNE     .next

    .loop:
        LDY     #$00
        LDA     (SAVED_RET_ADDR),Y
        BEQ     .end
        JSR     PUT_CHAR

    .next:
        INC     SAVED_RET_ADDR
        BNE     .loop
        INC     SAVED_RET_ADDR+1
        BNE     .loop

    .end:
        LDA     SAVED_RET_ADDR+1
        PHA
        LDA     SAVED_RET_ADDR
        PHA
        RTS
```

Defines:
PUT_STRING, used in chunks 53, 73a, 120, 121, 237, 241, 244, 259, 261–64, and 268.
Uses PUT_CHAR 46a and SAVED_RET_ADDR 46b.

Like PUT_CHAR, we also have PUT_DIGIT which draws the sprite corresponding to digits 0 to 9 at the current position, incrementing the cursor.

48a ⟨*put digit* 48a⟩≡ (278)

```
        ORG     $7B15
    PUT_DIGIT:
        SUBROUTINE
        ; Enter routine with A set to the digit to put on the screen.

        CLC
        ADC     #$3B                    ; '0' -> sprite 59, '9' -> sprite 68.
        LDX     DRAW_PAGE
        CPX     #$40
        BEQ     .draw_to_page2
        JSR     DRAW_SPRITE_PAGE1
        INC     GAME_COLNUM
        RTS

    .draw_to_page2:
        JSR     DRAW_SPRITE_PAGE2
        INC     GAME_COLNUM
        RTS
```

Defines:
PUT_DIGIT, used in chunks 51, 53, 74, and 120–22.
Uses DRAW_PAGE 45, DRAW_SPRITE_PAGE1 35, DRAW_SPRITE_PAGE2 35, and GAME_COLNUM 34a.

3.6 Numbers

We also need a way to put numbers on the screen.

First, a routine to convert a one-byte decimal number into hundreds, tens, and units.

48b ⟨*defines* 4⟩+≡ (281) ◁46b 50b▷

```
    HUNDREDS        EQU     $89
    TENS            EQU     $8A
    UNITS           EQU     $8B
```

Defines:
HUNDREDS, used in chunks 49, 53, 74, and 121c.
TENS, used in chunks 49–51, 53, 74, 121c, and 122a.
UNITS, used in chunks 49–51, 53, 74, 121c, and 122a.

49 ⟨*to decimal3* 49⟩≡ (278)

```
      ORG     $7AF8
  TO_DECIMAL3:
      SUBROUTINE
      ; Enter routine with A set to the number to convert.

      LDX     #$00
      STX     TENS
      STX     HUNDREDS

  .loop1:
      CMP     #100
      BCC     .loop2
      INC     HUNDREDS
      SBC     #100
      BNE     .loop1

  .loop2:
      CMP     #10
      BCC     .end
      INC     TENS
      SBC     #10
      BNE     .loop2

  .end:
      STA     UNITS
      RTS
```

Defines:
TO_DECIMAL3, used in chunks 53, 74, and 121c.
Uses HUNDREDS 48b, TENS 48b, and UNITS 48b.

There's also a routine to convert a BCD byte to tens and units.

50a ⟨*bcd to decimal2* 50a⟩≡ (278)

```
        ORG     $7AE9
  BCD_TO_DECIMAL2:
        SUBROUTINE
        ; Enter routine with A set to the BCD number to convert.

        STA     TENS
        AND     #$0F
        STA     UNITS
        LDA     TENS
        LSR
        LSR
        LSR
        LSR
        STA     TENS
        RTS
```

Defines:
  BCD_TO_DECIMAL2, used in chunks 51 and 122a.
Uses TENS 48b and UNITS 48b.

3.7 Score and status

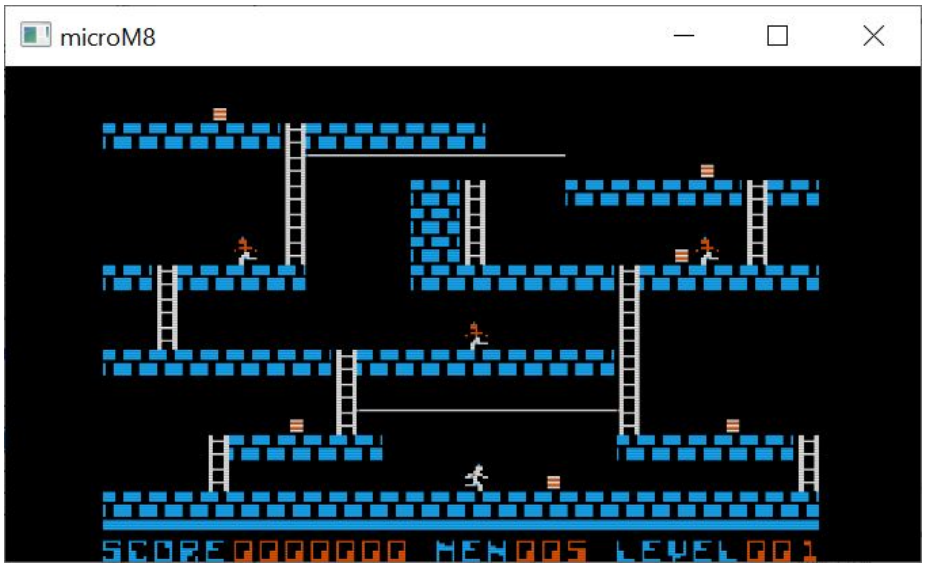
Lode Runner stores your score as an 8-digit BCD number.

50b ⟨*defines* 4⟩+≡ (281) ◁48b 52▷

```
  SCORE       EQU     $8E     ; 4 bytes, BCD format, tens/units in first byte.
```

Defines:
  SCORE, used in chunks 51, 53, 120a, 126, 137, 193, 233, 244, 246, 250, and 259.

The score is always put on the screen at row 16 column 5, but only the last 7 digits. Row 16 is the status line, as can be seen at the bottom of this screenshot.



There's a routine to add a 4-digit BCD number to the score and then update it on the screen.

51 ⟨*add and update score* 51⟩≡ (278)

```
    ORG     $7A92
ADD_AND_UPDATE_SCORE:
    SUBROUTINE
    ; Enter routine with A set to BCD tens/units and
    ; Y set to BCD thousands/hundreds.

    CLC
    SED                         ; Turn on BCD addition mode.
    ADC     SCORE
    STA     SCORE
    TYA
    ADC     SCORE+1
    STA     SCORE+1
    LDA     #$00
    ADC     SCORE+2
    STA     SCORE+2
    LDA     #$00
    ADC     SCORE+3
    STA     SCORE+3             ; SCORE += param
    CLD                         ; Turn off BCD addition mode.

    LDA     #5
    STA     GAME_COLNUM
```

```
        LDA     #16
        STA     GAME_ROWNUM

        LDA     SCORE+3
        JSR     BCD_TO_DECIMAL2
        LDA     UNITS               ; Note we skipped TENS.
        JSR     PUT_DIGIT

        LDA     SCORE+2
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT

        LDA     SCORE+1
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT

        LDA     SCORE
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JMP     PUT_DIGIT           ; tail call
```

Defines:
ADD_AND_UPDATE_SCORE, used in chunks 53, 126, 137, 193, and 250.
Uses BCD_TO_DECIMAL2 50a, GAME_COLNUM 34a, GAME_ROWNUM 34a, PUT_DIGIT 48a, SCORE 50b, TENS 48b, and UNITS 48b.

The other elements in the status line are the number of men (i.e. lives) and the current level.

52 ⟨*defines* 4⟩+≡ (281) ◁50b 57▷

```
LEVELNUM    EQU     $A6
LIVES       EQU     $98
```

Defines:
LEVELNUM, used in chunks 53, 74, 109b, 132b, 133c, 140a, 233, 250, and 268.
LIVES, used in chunks 53, 140, 141, 148, 246, 250, and 259.

Here are the routines to put the lives and level number on the status line. Lives starts at column 16, and level number starts at column 25.

53 ⟨*put status* 53⟩≡ (278)

```
        ORG     $7A70
    PUT_STATUS_LIVES:
        SUBROUTINE

        LDA     LIVES
        LDX     #16
        ; fallthrough

    PUT_STATUS_BYTE:
        SUBROUTINE
        ; Puts the number in A as a three-digit decimal on the screen
        ; at row 16, column X.

        STX     GAME_COLNUM
        JSR     TO_DECIMAL3
        LDA     #16
        STA     GAME_ROWNUM
        LDA     HUNDREDS
        JSR     PUT_DIGIT
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JMP     PUT_DIGIT           ; tail call

    PUT_STATUS_LEVEL:
        SUBROUTINE

        LDA     LEVELNUM
        LDX     #25
        BNE     PUT_STATUS_BYTE     ; Unconditional jump

        ORG     $79AD
    PUT_STATUS:
        SUBROUTINE

        JSR     CLEAR_HGR1
        JSR     CLEAR_HGR2

    PUT_STATUS_DRAW:
        LDY     #$27
        LDA     DRAW_PAGE
        CMP     #$40
        BEQ     .draw_line_on_page_2

    .draw_line_on_page_1:
        LDA     #$AA
```

```
    STA     $2350,Y
    STA     $2750,Y
    STA     $2B50,Y
    STA     $2F50,Y
    DEY
    LDA     #$D5
    STA     $2350,Y
    STA     $2750,Y
    STA     $2B50,Y
    STA     $2F50,Y
    DEY
    BPL     .draw_line_on_page_1
    BMI     .end        ; Unconditional

.draw_line_on_page_2:
    LDA     #$AA
    STA     $4350,Y
    STA     $4750,Y
    STA     $4B50,Y
    STA     $4F50,Y
    DEY
    LDA     #$D5
    STA     $4350,Y
    STA     $4750,Y
    STA     $4B50,Y
    STA     $4F50,Y
    DEY
    BPL     .draw_line_on_page_2

.end:
    LDA     #$10
    STA     GAME_ROWNUM
    LDA     #$00
    STA     GAME_COLNUM

    ; "SCORE        MEN    LEVEL   "
    JSR     PUT_STRING
    HEX     D3 C3 CF D2 C5 A0 A0 A0 A0 A0 A0 A0 A0 CD C5 CE
    HEX     A0 A0 A0 A0 CC C5 D6 C5 CC A0 A0 A0 00

    JSR     PUT_STATUS_LIVES
    JSR     PUT_STATUS_LEVEL
    LDA     #$00
    TAY
    JMP     ADD_AND_UPDATE_SCORE        ; tailcall
```

Defines:
PUT_STATUS, used in chunk 246.
PUT_STATUS_LEVEL, used in chunk 89.
PUT_STATUS_LIVES, used in chunks 89, 140b, and 250.

Uses ADD_AND_UPDATE_SCORE 51, CLEAR_HGR1 5, CLEAR_HGR2 5, DRAW_PAGE 45, GAME_COLNUM 34a, GAME_ROWNUM 34a, HUNDREDS 48b, LEVELNUM 52, LIVES 52, PUT_DIGIT 48a, PUT_STRING 47, SCORE 50b, TENS 48b, TO_DECIMAL3 49, and UNITS 48b.

Chapter 4

Sound

4.1 Simple beep

This simple beep routine clicks the speaker every 656 cycles. At approximately 980 nsec per cycle, this would be a period of about 0.64 milliseconds, or a tone of 1.56 kHz. This is a short beep, playing for a little over 0.1 seconds.

56 ⟨*beep* 56⟩≡ (278)

```
        ORG     $86CE
    BEEP:
        SUBROUTINE

        LDY     #$C0

    .loop:
        ; From here to click is 651 cycles. Additional 5 cycles afterwards.
        LDX     #$80            ; 2 cycles

        ; delay 640 cycles
    .loop2:
        DEX                     ; 2 cycles
        BNE     .loop2          ; 3 cycles

        LDA     ENABLE_SOUND    ; 3 cycles
        BEQ     .next           ; 3 cycles
        LDA     SPKR            ; 3 cycles

    .next:
        DEY                     ; 2 cycles
        BNE     .loop           ; 3 cycles
        RTS
```

Defines:
BEEP, used in chunks 73a, 74, 233, 259, 265, and 268.
Uses ENABLE_SOUND 59b and SPKR 59b.

4.2 Sound "strings"

A sound "string" describes a sound to play in terms of pitch and duration, ending in a `00`. Just like in the `PUT_STRING` routine, rather than take an address pointing to a sound string, instead it uses the return address as the source for data. It then has to fix up the actual return address at the end to be just after the zero-terminating byte of the string.

Because `NOTE_INDEX` is not zeroed out, this actually appends to the sound data buffer.

The format of a sound string is duration, followed by pitch, although the pitch is lower for higher numbers.

One example of a sound string is `07 45 06 55 05 44 04 54 03 43 02 53`, found in `CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER`.

57 ⟨*defines* 4⟩+≡ (281) ◁52 59b▷

```
NOTE_INDEX      EQU     $54
SOUND_DURATION  EQU     $0E00       ; 128 bytes
SOUND_PITCH     EQU     $0E80       ; 128 bytes
```

Defines:
`NOTE_INDEX`, used in chunks 58, 59a, 62, and 250.
`SOUND_DURATION`, used in chunks 58, 59a, and 62.
`SOUND_PITCH`, used in chunks 58, 59a, and 62.

58 ⟨*load sound data* 58⟩≡ (278)

```
      ORG     $87E1
  LOAD_SOUND_DATA:
      SUBROUTINE

      PLA
      STA     SAVED_RET_ADDR
      PLA
      STA     SAVED_RET_ADDR+1
      BNE     .next

  .loop:
      LDY     #$00
      LDA     (SAVED_RET_ADDR),Y
      BEQ     .end
      INC     NOTE_INDEX
      LDX     NOTE_INDEX
      STA     SOUND_DURATION,X
      INY
      LDA     (SAVED_RET_ADDR),Y
      STA     SOUND_PITCH,X

      INC     SAVED_RET_ADDR
      BNE     .next
      INC     SAVED_RET_ADDR+1

  .next:
      INC     SAVED_RET_ADDR
      BNE     .loop
      INC     SAVED_RET_ADDR+1
      BNE     .loop

  .end:
      LDA     SAVED_RET_ADDR+1
      PHA
      LDA     SAVED_RET_ADDR
      PHA
      RTS
```

Defines:
  LOAD_SOUND_DATA, used in chunks 137, 188, 193, and 250.
Uses NOTE_INDEX 57, SAVED_RET_ADDR 46b, SOUND_DURATION 57, and SOUND_PITCH 57.

There's also a simple routine to append a single note to the sound buffer. The routine gets called with the pitch in `A` and the duration in `X`.

59a ⟨*append note* 59a⟩≡ (278)

```
      ORG     $87D5
  APPEND_NOTE:
      SUBROUTINE

      INC     NOTE_INDEX
      LDY     NOTE_INDEX
      STA     SOUND_PITCH,Y
      TXA
      STA     SOUND_DURATION,Y
      RTS
```

Defines:
`APPEND_NOTE`, used in chunks 63, 168, and 171.
Uses `NOTE_INDEX` 57, `SOUND_DURATION` 57, and `SOUND_PITCH` 57.

4.3 Playing notes

The `PLAY_NOTE` routines plays a note through the built-in speaker. The time the note is played is based on `X` and `Y` forming a 16-bit counter (`X` being the most significant byte), but `A` controls the pitch, which is how often the speaker is clicked. The higher `A`, the lower the pitch.

The `ENABLE_SOUND` location can also disable playing the note, but the routine still takes as long as it would have.

59b ⟨*defines* 4⟩+≡ (281) ◁57 61b▷

```
  ENABLE_SOUND    EQU     $99     ; If 0, do not click speaker.
  SPKR            EQU     $C030   ; Access clicks the speaker.
```

Defines:
`ENABLE_SOUND`, used in chunks 56, 60a, 133c, and 142a.
`SPKR`, used in chunks 56 and 60a.

60a ⟨*play note* 60a⟩≡ (278)

```
        ORG     $87BA
    PLAY_NOTE:
        SUBROUTINE

        STA     TMP_PTR
        STX     TMP_PTR+1

    .loop:
        LDA     ENABLE_SOUND
        BEQ     .decrement_counter
        LDA     SPKR

    .decrement_counter:
        DEY
        BNE     .counter_decremented
        DEC     TMP_PTR+1
        BEQ     .end

    .counter_decremented:
        DEX
        BNE     .decrement_counter
        LDX     TMP_PTR
        JMP     .loop

    .end:
        RTS
```

Defines:
PLAY_NOTE, used in chunks 62 and 175.
Uses ENABLE_SOUND 59b, SPKR 59b, and TMP_PTR 4.

4.4 Playing a sound

The SOUND_DELAY routine delays an amount of time based on the X register. The total number of cycles is about 905 per each X. Since the Apple //e clock cycle was 980 nsec (on an NTSC system), this routine would delay approximately 887 microseconds times X. PAL systems were very slightly slower (by 0.47%), which corresponds to 883 microseconds times X.

60b ⟨*tables* 9⟩+≡ (281) ◁34b 73b▷

```
        ORG     $86BE
    SOUND_DELAY_AMOUNTS:
        HEX     02 04 06 08 0A 0C 0E 10 12 14 16 18 1A 1C 1E 20
```

Defines:
SOUND_DELAY_AMOUNTS, used in chunk 61a.

61a ⟨*sound delay* 61a⟩≡ (278)

```
      ORG     $86B1
  SOUND_DELAY:
      SUBROUTINE

      LDA     SOUND_DELAY_AMOUNTS,X
      TAX

  SOUND_DELAY1:
      LDY     #$B4          ; 180
  .loop:
      DEY                   ; 2 cycles
      BNE     .loop         ; 3 cycles
      DEX                   ; 2 cycles
      BNE     SOUND_DELAY1 ; 3 cycles
      RTS
```

Defines:
SOUND_DELAY, used in chunk 62.
SOUND_DELAY1, used in chunk 62.
Uses SOUND_DELAY_AMOUNTS 60b.

Finally, the PLAY_SOUND routine plays one section of the sound string stored in the SOUND_PITCH and SOUND_DURATION buffers. We have to break up the playing of the sound so that gameplay doesn't pause while playing the sound, although game play does pause while playing the note.

Alternatively, if there is no sound string, we can play the note stored in location \$A4 as long as location \$9B is zero. The duration is 2 + FRAME_PERIOD.

The routine is designed to delay approximately the same amount regardless of sound duration. The delay is controlled by FRAME_PERIOD. This value is hardcoded to 6 initially, but the game can be sped up, slowed down, or even paused.

61b ⟨*defines* 4⟩+≡ (281) ◁59b 64▷

```
  FRAME_PERIOD    EQU     $8C     ; initially 6
```

Defines:
FRAME_PERIOD, used in chunks 62 and 143.

62 ⟨*play sound* 62⟩≡ (278)

```
        ORG     $8811
    PLAY_SOUND:
        SUBROUTINE

        LDY     NOTE_INDEX
        BEQ     .no_more_notes
        LDA     SOUND_PITCH,Y
        LDX     SOUND_DURATION,Y
        JSR     PLAY_NOTE

        LDY     NOTE_INDEX              ; Y = NOTE_INDEX
        DEC     NOTE_INDEX              ; NOTE_INDEX--
        LDA     FRAME_PERIOD
        SEC
        SBC     SOUND_DURATION,Y        ; A = FRAME_PERIOD - SOUND_DURATION[Y]
        BEQ     .done
        BCC     .done                   ; If A <= 0, done.
        TAX
        JSR     SOUND_DELAY1

    .done:
        SEC
        RTS

    .no_more_notes:
        LDA     $9B
        BNE     .end
        LDA     $A4
        LSR                     ; pitch = $A4 >> 1
        INC     $A4             ; $A4++
        LDX     FRAME_PERIOD
        INX
        INX                     ; duration = FRAME_PERIOD + 2
        JSR     PLAY_NOTE

        CLC
        RTS

    .end:
        LDX     FRAME_PERIOD
        JSR     SOUND_DELAY

        CLC
        RTS
```

Defines:
PLAY_SOUND, used in chunks 63 and 250.
Uses FRAME_PERIOD 61b, NOTE_INDEX 57, PLAY_NOTE 60a, SOUND_DELAY 61a, SOUND_DELAY1 61a, SOUND_DURATION 57, and SOUND_PITCH 57.

Another routine is just for when a level is cleared. It appends a note based on a scratch location, and then plays it.

63 ⟨*append level cleared note* 63⟩≡ (278)

```
    ORG     $622A
APPEND_LEVEL_CLEARED_NOTE:
    SUBROUTINE

    LDA     SCRATCH_5C
    ASL
    ASL
    ASL
    ASL                         ; pitch = SCRATCH_5C * 16
    LDX     #$06                ; duration
    JSR     APPEND_NOTE
    JMP     PLAY_SOUND
```

Defines:
APPEND_LEVEL_CLEARED_NOTE, used in chunk 250.
Uses APPEND_NOTE 59a, PLAY_SOUND 62, and SCRATCH_5C 4.

Chapter 5

Input

5.1 Joystick input

Analog joysticks (or paddles) on the Apple //e are just variable resistors. The resistor on a paddle creates an RC circuit with a capacitor which can be discharged by accessing the `PTRIG` location. Once that is done, the capacitor starts charging through the resistor. The lower the resistor value, the faster the charge.

At the start, each `PADDL` value has its high bit set to one. When the voltage on the capacitor reaches `2/3` of the supply voltage, the corresponding `PADDL` switch will have its high bit set to zero. So, we just need to watch the `PADDL` value until it is non-negative, counting the amount of time it takes for that to happen.

In the `READ_PADDLES` routine, we trigger the paddles and then alternately read `PADDL0` and `PADDL1` until one of them indicates the threshold was reached. If the `PADDL` value hasn't yet triggered, we increment the corresponding `PADDLE_VALUE` location.

Once a `PADDL` triggers, we stop incrementing the corresponding `PADDLE_VALUE`.

Once both `PADDL` have been triggered, we end the routine.

64 ⟨*defines* 4⟩+≡ (281) ◁61b 66▷

```
PADDLE0_VALUE       EQU     $65
PADDLE1_VALUE       EQU     $66
PADDL0              EQU     $C064
PADDL1              EQU     $C065
PTRIG               EQU     $C070
```

Defines:
`PADDL0`, used in chunks 65 and 68a.
`PADDL1`, used in chunks 65 and 68a.
`PADDLE0_VALUE`, used in chunks 65, 67, and 151.
`PADDLE1_VALUE`, used in chunks 65, 67, and 151.

65 ⟨*read paddles* 65⟩≡ (278)

```
      ORG     $8746
  READ_PADDLES:
      SUBROUTINE

      LDA     #$00
      STA     PADDLE0_VALUE
      STA     PADDLE1_VALUE       ; Zero out values
      LDA     PTRIG

  .loop:
      LDX     #$01                ; Start with paddle 1

  .check_paddle:
      LDA     PADDL0,X
      BPL     .threshold_reached
      INC     PADDLE0_VALUE,X
  .check_next_paddle
      DEX
      BPL     .check_paddle

      ; Checked both paddles
      LDA     PADDL0
      ORA     PADDL1
      BPL     .end                ; Both paddles triggered, then end.
      LDA     PADDLE0_VALUE
      ORA     PADDLE1_VALUE
      BPL     .loop               ; Unconditional

  .threshold_reached:
      NOP
      BPL     .check_next_paddle      ; Unconditional

  .end:
      RTS
```

Defines:
READ_PADDLES, used in chunks 67 and 151.
Uses PADDL0 64, PADDL1 64, PADDLE0_VALUE 64, and PADDLE1_VALUE 64.

The INPUT_MODE location tells whether the player is using keyboard or joystick input.

The CHECK_JOYSTICK_OR_DELAY routine, if we are in joystick mode, reads the paddle values and checks to see if any value is below 0x12 or above 0x3A, and if so, declares that a paddle has a large enough input by setting the carry flag and returning.

If neither paddle has a large enough input, we also check the paddle buttons, and if either one is triggered, we set the carry and return.

Otherwise, if no paddle input was detected, or we're in keyboard mode, we clear the carry and return.

66 ⟨*defines* 4⟩+≡ (281) ◁64 68b▷

```
INPUT_MODE  EQU     $95         ; 0xCA = Joystick mode (J), 0xCB = Keyboard mode (K)
                                ; initially set to 0xCA
JOYSTICK_MODE   EQU     #$CA
KEYBOARD_MODE   EQU     #$CB

BUTN0       EQU     $C061       ; Or open apple
BUTN1       EQU     $C062       ; Or solid apple
```

Defines:
BUTN0, used in chunks 67, 131b, 145, 148, 151, and 256.
BUTN1, used in chunks 67, 131b, 145, 148, 151, and 256.
INPUT_MODE, used in chunks 67, 68a, 131b, 139, 142b, 145, 148, 256, 259, and 263.

67 ⟨*check joystick or delay* 67⟩≡ (278)

```
        ORG     $876D
    CHECK_JOYSTICK_OR_DELAY:
        SUBROUTINE

        LDA     INPUT_MODE
        CMP     #KEYBOARD_MODE
        BEQ     .delay_and_return       ; Keyboard mode, so just delay and return

        JSR     READ_PADDLES

        LDA     PADDLE0_VALUE
        CMP     #$12
        BCC     .have_joystick_input           ; PADDLE0_VALUE < 0x12
        CMP     #$3B
        BCS     .have_joystick_input           ; PADDLE0_VALUE >= 0x3B

        LDA     PADDLE1_VALUE
        CMP     #$12
        BCC     .have_joystick_input
        CMP     #$3B
        BCS     .have_joystick_input

        LDA     BUTN1
        BMI     .have_joystick_input
        LDA     BUTN0
        BMI     .have_joystick_input

        CLC
        RTS

    .have_joystick_input:
        SEC
        RTS

    .delay_and_return:
        LDX     #$02
    .loop:
        DEY
        BNE     .loop
        DEX
        BNE     .loop
        CLC
        RTS
```

Defines:
CHECK_JOYSTICK_OR_DELAY, used in chunks 70 and 71.
Uses BUTN0 66, BUTN1 66, INPUT_MODE 66, PADDLE0_VALUE 64, PADDLE1_VALUE 64, and READ_PADDLES 65.

If, after a timeout, a button hasn't been pressed, just assume keyboard mode.

68a ⟨*detect lack of joystick* 68a⟩≡ (278)

```
        ORG     $87A2
  DETECT_LACK_OF_JOYSTICK:
        SUBROUTINE

        LDA     PTRIG
        LDX     #$10

  .loop:
        LDA     PADDL0
        ORA     PADDL1
        BPL     .return

        DEY
        BNE     .loop
        DEX
        BNE     .loop

        LDA     #KEYBOARD_MODE
        STA     INPUT_MODE

  .return:
        RTS
```

Defines:
  DETECT_LACK_OF_JOYSTICK, used in chunk 277a.
Uses INPUT_MODE 66, PADDL0 64, and PADDL1 64.

5.2 Keyboard routines

The WAIT_KEY routine accesses the keyboard strobe softswitch KBDSTRB, which clears the keyboard strobe in readiness to get a key. When a key is pressed after the keyboard strobe is cleared, the key (with the high bit set) is accessible through KBD

68b ⟨*defines* 4⟩+≡ (281) ◁66 69c▷

```
  KBD           EQU     $C000
  KBDSTRB       EQU     $C010
```

Defines:
  KBD, used in chunks 69–71, 132a, 133c, 139, 145, 148, and 256.
  KBDSTRB, used in chunks 69, 72–74, 131a, 132c, 139, 145, 233, 256, 266, and 268.

69a ⟨*wait key* 69a⟩≡ (278)

```
        ORG     $869F
  WAIT_KEY:
        SUBROUTINE

        STA     KBDSTRB
        LDA     KBD
        BMI     WAIT_KEY
        RTS
```

Defines:
  WAIT_KEY, used in chunks 134d and 250.
Uses KBD 68b and KBDSTRB 68b.

The WAIT_KEY_QUEUED routine does not clear the keyboard strobe first, so if a key had been pressed before entering the routine, the routine will immediately return.

69b ⟨*wait key queued* 69b⟩≡ (278)

```
        ORG     $86A8
  WAIT_KEY_QUEUED:
        SUBROUTINE

        LDA     KBD
        BPL     WAIT_KEY_QUEUED
        STA     KBDSTRB
        RTS
```

Defines:
  WAIT_KEY_QUEUED, used in chunk 141b.
Uses KBD 68b and KBDSTRB 68b.

69c ⟨*defines* 4⟩+≡ (281) ◁68b 78b▷

```
        ORG     $8745
  CURSOR_SPRITE:
        HEX     06
```

Defines:
  CURSOR_SPRITE, used in chunks 70 and 71.

70 ⟨*wait for key* 70⟩≡ (278)

```
        ORG     $85F3
    WAIT_FOR_KEY:
        SUBROUTINE
        ; Enter routine with A set to cursor sprite. If zero, sprite 10 (all white)
        ; will be used.

        STA     CURSOR_SPRITE

    .loop:
        LDA     #$68
        STA     SCRATCH_A1
        LDA     CURSOR_SPRITE
        BNE     .draw_sprite
        LDA     #SPRITE_ALLWHITE
    .draw_sprite:
        JSR     DRAW_SPRITE_PAGE2

    .loop2:
        LDA     KBD
        BMI     .end            ; on keypress, end

        JSR     CHECK_JOYSTICK_OR_DELAY
        DEC     SCRATCH_A1
        BNE     .loop2

        ; Draw a blank
        LDA     #$00
        JSR     DRAW_SPRITE_PAGE2
        LDA     #$68
        STA     SCRATCH_A1

    .loop3:
        LDA     KBD
        BMI     .end
        JSR     CHECK_JOYSTICK_OR_DELAY
        DEC     SCRATCH_A1
        BNE     .loop3
        JMP     .loop

    .end:
        PHA
        LDA     CURSOR_SPRITE
        JSR     DRAW_SPRITE_PAGE2
        PLA
        RTS
```

Defines:
WAIT_FOR_KEY, used in chunks 73a, 233, and 268.
Uses CHECK_JOYSTICK_OR_DELAY 67, CURSOR_SPRITE 69c, DRAW_SPRITE_PAGE2 35, KBD 68b, and SCRATCH_A1 4.

71 ⟨*wait for key page1* 71⟩≡ (278)

```
        ORG     $8700
    WAIT_FOR_KEY_WITH_CURSOR_PAGE_1:
        SUBROUTINE
        ; Enter routine with A set to cursor sprite. If zero, sprite 10 (all white)
        ; will be used.

        STA     CURSOR_SPRITE

    .loop:
        LDA     #$68
        STA     SCRATCH_A1
        LDA     #$00
        LDX     CURSOR_SPRITE
        BNE     .draw_sprite
        LDA     #SPRITE_ALLWHITE
    .draw_sprite:
        JSR     DRAW_SPRITE_PAGE1

    .loop2:
        LDA     KBD
        BMI     .end            ; on keypress, end

        JSR     CHECK_JOYSTICK_OR_DELAY
        BCS     .end

        DEC     SCRATCH_A1
        BNE     .loop2

        LDA     CURSOR_SPRITE
        JSR     DRAW_SPRITE_PAGE1
        LDA     #$68
        STA     SCRATCH_A1

    .loop3:
        LDA     KBD
        BMI     .end

        JSR     CHECK_JOYSTICK_OR_DELAY
        BCS     .end

        DEC     SCRATCH_A1
        BNE     .loop3
        JMP     .loop

    .end:
        PHA
        LDA     CURSOR_SPRITE
        JSR     DRAW_SPRITE_PAGE1
        PLA
```

```
    RTS
```

Defines:
WAIT_FOR_KEY_WITH_CURSOR_PAGE_1, used in chunks 72, 74, 250, and 266.
Uses CHECK_JOYSTICK_OR_DELAY 67, CURSOR_SPRITE 69c, DRAW_SPRITE_PAGE1 35, KBD 68b, and SCRATCH_A1 4.

This routine is used by the level editor whenever we need to wait for a key. If the key isn't the escape key, we can immediately exit, and the caller interprets the key. However, on escape, we abort whatever editor command we were in the middle of, and just go back to the main editor command loop, asking for an editor command.

72 ⟨*editor wait for key* 72⟩≡ (278)

```
    ORG     $823D
EDITOR_WAIT_FOR_KEY:
    SUBROUTINE

    LDA     #$00
    JSR     WAIT_FOR_KEY_WITH_CURSOR_PAGE_1
    STA     KBDSTRB
    CMP     #$9B        ; ESC
    BNE     .return
    JMP     EDITOR_COMMAND_LOOP

.return
    RTS
```

Defines:
EDITOR_WAIT_FOR_KEY, used in chunks 241, 244, 259, and 262.
Uses EDITOR_COMMAND_LOOP 259, KBDSTRB 68b, and WAIT_FOR_KEY_WITH_CURSOR_PAGE_1 71.

73a ⟨*hit key to continue* 73a⟩≡ (278)

```
      ORG     $80D8
  HIT_KEY_TO_CONTINUE:
      SUBROUTINE

      ; "\r"
      ; "\r"
      ; "HIT A KEY TO CONTINUE "
      JSR     PUT_STRING
      HEX     8D 8D C8 C9 D4 A0 C1 A0 CB C5 D9 A0 D4 CF A0 C3
      HEX     CF CE D4 C9 CE D5 C5 A0 00

      JSR     BEEP
      STA     TXTPAGE2
      LDA     #$00
      JSR     WAIT_FOR_KEY
      STA     KBDSTRB
      STA     TXTPAGE1
  RETURN_FROM_SUBROUTINE:
      RTS
```

Defines:
HIT_KEY_TO_CONTINUE, used in chunk 237.
RETURN_FROM_SUBROUTINE, used in chunk 238.
Uses BEEP 56, KBDSTRB 68b, PUT_STRING 47, TXTPAGE1 130a, TXTPAGE2 122c, and WAIT_FOR_KEY 70.

The GET_LEVEL_FROM_KEYBOARD is used by the level editor to ask the user for a 3-digit level number. The current level number, given by DISK_LEVEL_LOC, is put on the screen. Note that DISK_LEVEL_LOC is 0-based, while the levels the user enters are 1-based, so there's an increment at the beginning and a decrement at the end.

The routine handles forward and backward arrows. Hitting the escape key aborts the editor action and dumps the user back into the editor command loop. Hitting the return key accepts the user's input, and the level is stored in DISK_LEVEL_LOC and LEVELNUM.

73b ⟨*tables* 9⟩+≡ (281) ◁60b 78a▷

```
      ORG     $824E
  SAVED_GAME_COLNUM:
      HEX     85
```

Defines:
SAVED_GAME_COLNUM, used in chunk 74.

74 ⟨*get level from keyboard* 74⟩≡ (278)

```
      ORG     $817B
  GET_LEVEL_FROM_KEYBOARD:
      SUBROUTINE

      LDY     DISK_LEVEL_LOC
      INY
      TYA
      JSR     TO_DECIMAL3     ; make 1-based
      LDA     GAME_COLNUM
      STA     SAVED_GAME_COLNUM
      LDY     #$00

      ; Print current level
  .loop:
      LDA     HUNDREDS,Y
      STY     KBD_ENTRY_INDEX     ; save Y
      JSR     PUT_DIGIT
      LDY     KBD_ENTRY_INDEX     ; restore Y
      INY
      CPY     #$03
      BCC     .loop

      LDA     SAVED_GAME_COLNUM
      STA     GAME_COLNUM
      LDY     #$00
      STY     KBD_ENTRY_INDEX

  .loop2
      LDX     KBD_ENTRY_INDEX
      LDA     HUNDREDS,X
      CLC
      ADC     #$3B            ; sprite = '0' + X
      JSR     WAIT_FOR_KEY_WITH_CURSOR_PAGE_1
      STA     KBDSTRB
      CMP     #$8D            ; return
      BEQ     .return_pressed

      CMP     #$88            ; backspace
      BNE     .check_for_fwd_arrow

      LDX     KBD_ENTRY_INDEX
      BEQ     .beep           ; can't backspace past the beginning

      DEC     KBD_ENTRY_INDEX
      DEC     GAME_COLNUM
      JMP     .loop2

  .check_for_fwd_arrow:
      CMP     #$95            ; fwd arrow
```

```
    BNE     .check_for_escape

    LDX     KBD_ENTRY_INDEX
    CPX     #$02
    BEQ     .beep           ; can't fwd past the end

    INC     GAME_COLNUM
    INC     KBD_ENTRY_INDEX
    JMP     .loop2

.check_for_escape:
    CMP     #$9B            ; ESC
    BNE     .check_for_digit
    JMP     EDITOR_COMMAND_LOOP

.check_for_digit:
    CMP     #$B0            ; '0'
    BCC     .beep           ; less than '0' not allowed
    CMP     #$BA            ; '9'+1
    BCS     .beep           ; greater than '9' not allowed

    SEC
    SBC     #$B0            ; char - '0'
    LDY     KBD_ENTRY_INDEX
    STA     HUNDREDS,Y
    JSR     PUT_DIGIT
    INC     KBD_ENTRY_INDEX
    LDA     KBD_ENTRY_INDEX
    CMP     #$03
    BCC     .loop2

    ; Don't allow a fourth digit
    DEC     KBD_ENTRY_INDEX
    DEC     GAME_COLNUM
    JMP     .loop2

.beep:
    JSR     BEEP
    JMP     .loop2

.return_pressed:
    LDA     SAVED_GAME_COLNUM
    CLC
    ADC     #$03
    STA     GAME_COLNUM
    LDA     #$00
    LDX     HUNDREDS
    BEQ     .add_tens

    CLC
```

```
.loop_hundreds:
    ADC     #100
    BCS     .end
    DEX
    BNE     .loop_hundreds

.add_tens:
    LDX     TENS
    BEQ     .add_units

    CLC
.loop_tens:
    ADC     #10
    BCS     .end
    DEX
    BNE     .loop_tens

.add_units:
    CLC
    ADC     UNITS
    BCS     .end

    STA     LEVELNUM
    TAY
    DEY
    STY     DISK_LEVEL_LOC
    CPY     #$96

.end:
    RTS
```

Defines:
GET_LEVEL_FROM_KEYBOARD, used in chunks 261–64.
Uses BEEP 56, EDITOR_COMMAND_LOOP 259, GAME_COLNUM 34a, HUNDREDS 48b, KBD_ENTRY_INDEX 233, KBDSTRB 68b, LEVELNUM 52, PUT_DIGIT 48a, SAVED_GAME_COLNUM 73b, TENS 48b, TO_DECIMAL3 49, UNITS 48b, and WAIT_FOR_KEY_WITH_CURSOR_PAGE_1 71.

Chapter 6

Levels

One of the appealing things about Lode Runner are its levels. 150 levels are stored in the game, and there is even a level editor included.

6.1 Drawing a level

Let's see how Lode Runner draws a level. We start with the routine `DRAW_LEVEL_PAGE2`, which draws a level on HGR2. Note that HGR1 would be displayed, so the player doesn't see the draw happening.

We start by looping backwards over rows 15 through 0:

77 ⟨*level draw routine* 77⟩≡ (278) 81a▷

```
        ORG     $63B3
    DRAW_LEVEL_PAGE2:
        SUBROUTINE
        ; Returns carry set if there was no player sprite in the level,
        ; or carry clear if there was.

        LDY     #MAX_GAME_ROW
        STY     GAME_ROWNUM

    .row_loop:
```

Defines:
`DRAW_LEVEL_PAGE2`, used in chunk 113.
Uses `GAME_ROWNUM` 34a.

We'll assume the level data is stored in a table which contains 16 pointers, one for each row. As usual in Lode Runner, the pages and offsets for those pointers are stored in separate tables. these are CURR_LEVEL_ROW_SPRITES_PTR_PAGES and CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS.

78a ⟨*tables* 9⟩+≡ (281) ◁73b 82a▷

```
    ORG     $1C05
CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS:
    HEX     00 1C 38 54 70 8C A8 C4 E0 FC 18 34 50 6C 88 A4
CURR_LEVEL_ROW_SPRITES_PTR_PAGES:
    HEX     08 08 08 08 08 08 08 08 08 08 09 09 09 09 09 09
CURR_LEVEL_ROW_SPRITES_PTR_PAGES2:
    HEX     0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0B 0B 0B 0B 0B 0B
```

Defines:
CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS, used in chunks 78–80 and 157.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES, used in chunks 78–80 and 157.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES2, used in chunks 78–80.

At the beginning of this loop, we create two pointers which we'll simply call PTR1 and PTR2.

78b ⟨*defines* 4⟩+≡ (281) ◁69c 80c▷

```
PTR1        EQU     $06     ; 2 bytes
PTR2        EQU     $08     ; 2 bytes
```

Defines:
PTR1, used in chunks 78–82, 87, 114, 116, 126, 157, 159, 161, 164, 168, 171, 174, 175, 179, 188, 193, 201, 204, 206, 208, 210, 265, and 266.
PTR2, used in chunks 78–80, 82–84, 114, 126, 137, 145, 157, 159, 161, 164, 175, 179, 184, 193, 198, 201, 204, 206, 208, 210, 212, 215, 217, 219, 221, and 224.

We set PTR1 to the pointer corresponding to the current row, and PTR2 to the other page, though I don't know what it's for yet, I think a "background" page that contains only non-moving elements.

These are very useful fragments, and appear all over the place in the code. This fragment sets PTR1 to the current active level's row sprite data.

78c ⟨*set active row pointer* PTR1 *for* Y 78c⟩≡ (80d 87 116 157 159 174 175 188 193 201 208 210 265 266)

```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES,Y
    STA     PTR1+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 78a, and PTR1 78b.

This fragment sets PTR2 to the current background level's row sprite data.

78d ⟨*set background row pointer* PTR2 *for* Y 78d⟩≡ (80e 126 137 145 175 184 193 198 212 215 217 219 221 224)

```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2,Y
    STA     PTR2+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 78a, and PTR2 78b.

And this fragment sets PTR1 to the active row and PTR2 to the background row.

79a ⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y 79a⟩≡ (79c 81a 113 126 157 159 161 164 175 179 193 204 206

```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR1
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES,Y
    STA     PTR1+1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2,Y
    STA     PTR2+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 78a, PTR1 78b, and PTR2 78b.

Occasionally the sets are reversed, although the effect is identical, so:

79b ⟨*set active and background row pointers* PTR2 *and* PTR1 *for* Y+1 79b⟩≡ (193)

```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
    STA     PTR1
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2+1,Y
    STA     PTR2+1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES+1,Y
    STA     PTR1+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 78a, PTR1 78b, and PTR2 78b.

There's even a routine which does this, but it seems that there was a lot of inlining instead. Presumably the cycles were more important than the space.

79c ⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y *routine* 79c⟩≡ (278)

```
    ORG     $884B
GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA:
    SUBROUTINE

    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    RTS
```

Defines:
GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA, used in chunks 168 and 171.

Occasionally we want to get the next row (i.e. for Y+1). In that case we use these fragments.

79d ⟨*set active row pointer* PTR1 *for* Y+1 79d⟩≡ (159)

```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
    STA     PTR1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES+1,Y
    STA     PTR1+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 78a, and PTR1 78b.

80a ⟨*set background row pointer* PTR2 *for* Y+1 80a⟩≡ (80f 198 201 215 217 219 221 224)

```
LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
STA     PTR2
LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2+1,Y
STA     PTR2+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 78a, and PTR2 78b.

80b ⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y+1 80b⟩≡ (175)

```
LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
STA     PTR1
STA     PTR2
LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES+1,Y
STA     PTR1+1
LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2+1,Y
STA     PTR2+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 78a, PTR1 78b, and PTR2 78b.

We also keep track of the player's sprite column and row.

80c ⟨*defines* 4⟩+≡ (281) ◁78b 81d▷

```
PLAYER_COL      EQU     $00
PLAYER_ROW      EQU     $01
```

Defines:
PLAYER_COL, used in chunks 80, 84c, 85c, 112b, 135b, 137, 157, 159, 161, 164, 168, 171, 175, 198, and 250.
PLAYER_ROW, used in chunks 80, 84c, 135b, 137, 157, 159, 161, 164, 168, 171, 174, 175, 198, 213, 221, 224, and 250.

A common paradigm is to get the sprite where the player is, on the active or background page, so these fragments are repeated many times:

80d ⟨*get active sprite at player location* 80d⟩≡

```
LDY     PLAYER_ROW
⟨set active row pointer PTR1 for Y 78c⟩
LDY     PLAYER_COL
LDA     (PTR1),Y
```

Uses PLAYER_COL 80c, PLAYER_ROW 80c, and PTR1 78b.

80e ⟨*get background sprite at player location* 80e⟩≡ (157)

```
LDY     PLAYER_ROW
⟨set background row pointer PTR2 for Y 78d⟩
LDY     PLAYER_COL
LDA     (PTR2),Y
```

Uses PLAYER_COL 80c, PLAYER_ROW 80c, and PTR2 78b.

80f ⟨*get background sprite at player location on next row* 80f⟩≡ (157)

```
LDY     PLAYER_ROW
⟨set background row pointer PTR2 for Y+1 80a⟩
LDY     PLAYER_COL
LDA     (PTR2),Y
```

Uses PLAYER_COL 80c, PLAYER_ROW 80c, and PTR2 78b.

81a ⟨*level draw routine* 77⟩+≡ (278) ◁77 81b▷
⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y 79a⟩

Next, we loop over the columns backwards from 27 to 0.

81b ⟨*level draw routine* 77⟩+≡ (278) ◁81a 81c▷
```
    LDY     #MAX_GAME_COL
    STY     GAME_COLNUM

.col_loop:
```
Uses GAME_COLNUM 34a.

We load the sprite from the level data.

81c ⟨*level draw routine* 77⟩+≡ (278) ◁81b 81f▷
```
    LDA     (PTR1),Y
```
Uses PTR1 78b.

Now, as we place each sprite, we count the number of each piece we've used so far. Remember that anyone can create a level, but there are some limitations. Specifically, we are limited to 45 ladders, one player, and 5 guards. We store the counts as we go.

These values are zeroed before the DRAW_LEVEL_PAGE2 routine is called.

81d ⟨*defines* 4⟩+≡ (281) ◁80c 81e▷
```
GUARD_COUNT     EQU     $8D
GOLD_COUNT      EQU     $93
LADDER_COUNT    EQU     $A3
```
Defines:
GOLD_COUNT, used in chunks 83a, 112b, 126, 137, 179, 193, and 250.
GUARD_COUNT, used in chunks 83b, 112b, 126, 145, 188, 191a, 193, and 250.
LADDER_COUNT, used in chunks 82b, 112b, and 179.

However, there's a flag called VERBATIM that tells us whether we want to ignore these counts and just draw the level as specified. Possibly when we're using the level editor.

81e ⟨*defines* 4⟩+≡ (281) ◁81d 84b▷
```
VERBATIM        EQU     $A2
```
Defines:
VERBATIM, used in chunks 81f, 85c, and 111b.

81f ⟨*level draw routine* 77⟩+≡ (278) ◁81c 82b▷
```
    LDX     VERBATIM
    BEQ     .draw_sprite1       ; This will then unconditionally jump to
                                ; .draw_sprite2. We have to do that because of
                                ; relative jump amount limitations.
```
Uses VERBATIM 81e.

Next we handle sprite 6, which is a symbol used to denote ladder placement. If we've already got the maximum number of ladders, we just put in a space instead. For each ladder placed, we write the LADDER_LOCS table with its coordinates.

82a ⟨*tables* 9⟩+≡ (281) ◁78a 99a▷

```
LADDER_LOCS_COL     EQU     $0C00   ; 48 bytes
LADDER_LOCS_ROW     EQU     $0C30   ; 48 bytes
```

Defines:
LADDER_LOCS_COL, used in chunks 82b and 179.
LADDER_LOCS_ROW, used in chunks 82b and 179.

82b ⟨*level draw routine* 77⟩+≡ (278) ◁81f 82c▷

```
    CMP     #SPRITE_INVISIBLE_LADDER
    BNE     .check_for_gold

    LDX     LADDER_COUNT
    CPX     #45
    BCS     .remove_sprite

    INC     LADDER_COUNT
    INX
    LDA     GAME_ROWNUM
    STA     LADDER_LOCS_ROW,X
    TYA
    STA     LADDER_LOCS_COL,X
```

Uses GAME_ROWNUM 34a, LADDER_COUNT 81d, LADDER_LOCS_COL 82a, and LADDER_LOCS_ROW 82a.

In any case, we remove the sprite from the current level data.

82c ⟨*level draw routine* 77⟩+≡ (278) ◁82b 83a▷

```
.remove_sprite:
    LDA     #SPRITE_EMPTY
    STA     (PTR1),Y
    STA     (PTR2),Y

.draw_sprite1
    BEQ     .draw_sprite        ; Unconditional jump.
```

Uses PTR1 78b and PTR2 78b.

Next, we check for sprite 7, the gold box.

83a ⟨*level draw routine* 77⟩+≡ (278) ◁82c 83b▷

```
.check_for_gold:
    CMP      #SPRITE_GOLD
    BNE      .check_for_guard

    INC      GOLD_COUNT
    BNE      .draw_sprite        ; This leads to a situation where if we wrap
                                 ; GOLD_COUNT around back to 0 (so 256 boxes)
                                 ; we end up falling through, which eventually
                                 ; just draws the sprite anyway. So this is kind
                                 ; of unconditional.
```

Uses GOLD_COUNT 81d.

Next, we check for sprite 8, a guard. If we've already got the maximum number of guards, we just put in a space instead. For each guard placed, we write the GUARD_LOCS table with its coordinates. We also write some other guard-related tables.

83b ⟨*level draw routine* 77⟩+≡ (278) ◁83a 84a▷

```
.check_for_guard:
    CMP      #SPRITE_GUARD
    BNE      .check_for_player

    LDX      GUARD_COUNT
    CPX      #5
    BCS      .remove_sprite          ; If GUARD_COUNT >= 5, remove sprite.

    INC      GUARD_COUNT
    INX
    TYA
    STA      GUARD_LOCS_COL,X
    LDA      GAME_ROWNUM
    STA      GUARD_LOCS_ROW,X
    LDA      #$00
    STA      GUARD_GOLD_TIMERS,X
    STA      GUARD_ANIM_STATES,X
    LDA      #$02
    STA      GUARD_X_ADJS,X
    STA      GUARD_Y_ADJS,X

    LDA      #SPRITE_EMPTY
    STA      (PTR2),Y
    LDA      #SPRITE_GUARD
    BNE      .draw_sprite            ; Unconditional jump.
```

Uses GAME_ROWNUM 34a, GUARD_ANIM_STATES 181, GUARD_COUNT 81d, GUARD_GOLD_TIMERS 181, GUARD_LOCS_COL 181, GUARD_LOCS_ROW 181, GUARD_X_ADJS 181, GUARD_Y_ADJS 181, and PTR2 78b.

Here we insert a few unconditional branches because of relative jump limitations.

84a ⟨*level draw routine* 77⟩+≡ (278) ◁83b 84c▷

```
.next_row:
    BPL     .row_loop
.next_col:
    BPL     .col_loop
```

Next we check for sprite 9, the player.

84b ⟨*defines* 4⟩+≡ (281) ◁81e 88▷

```
PLAYER_X_ADJ                EQU     $02     ; [0-4] minus 2 (so 2 = right on the sprite location)
PLAYER_Y_ADJ                EQU     $03     ; [0-4] minus 2 (so 2 = right on the sprite location)
PLAYER_ANIM_STATE           EQU     $04     ; Index into SPRITE_ANIM_SEQS
PLAYER_FACING_DIRECTION     EQU     $05     ; Hi bit set: facing left, otherwise facing right
```

Defines:
PLAYER_ANIM_STATE, used in chunks 84c, 135b, 136a, 168, 171, 175, and 185.
PLAYER_X_ADJ, used in chunks 84c, 135b, 137, 154, 161, and 164.
PLAYER_Y_ADJ, used in chunks 84c, 135b, 137, 154, 157, 159, 175, and 250.
Uses SPRITE_ANIM_SEQS 135a.

84c ⟨*level draw routine* 77⟩+≡ (278) ◁84a 85a▷

```
.check_for_player:
    CMP     #SPRITE_PLAYER
    BNE     .check_for_t_thing

    LDX     PLAYER_COL
    BPL     .remove_sprite          ; If PLAYER_COL > 0, remove sprite.

    STY     PLAYER_COL
    LDX     GAME_ROWNUM
    STX     PLAYER_ROW
    LDX     #$02
    STX     PLAYER_X_ADJ
    STX     PLAYER_Y_ADJ            ; Set Player X and Y movement to 0.
    LDX     #$08
    STX     PLAYER_ANIM_STATE       ; Corresponds to sprite 9 (see SPRITE_ANIM_SEQS)

    LDA     #SPRITE_EMPTY
    STA     (PTR2),Y
    LDA     #SPRITE_PLAYER
    BNE     .draw_sprite            ; Unconditional jump.
```

Uses GAME_ROWNUM 34a, PLAYER_ANIM_STATE 84b, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_X_ADJ 84b, PLAYER_Y_ADJ 84b, PTR2 78b, and SPRITE_ANIM_SEQS 135a.

Finally, we check for sprite 5, the t-thing, and replace it with a brick. If the sprite is anything else, we just draw it.

85a ⟨*level draw routine* 77⟩+≡ (278) ◁84c 85b▷

```
.check_for_t_thing:
    CMP     #SPRITE_TRAP
    BNE     .draw_sprite
    LDA     #SPRITE_BRICK

    ; fallthrough to .draw_sprite
```

We finally draw the sprite, on page 2, and advance the loop.

85b ⟨*level draw routine* 77⟩+≡ (278) ◁85a 85c▷

```
.draw_sprite:
    JSR     DRAW_SPRITE_PAGE2

    DEC     GAME_COLNUM
    LDY     GAME_COLNUM
    BPL     .next_col               ; Jumps to .col_loop

    DEC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    BPL     .next_row               ; Jumps to .row_loop
```

Uses DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, and GAME_ROWNUM 34a.

After the loop, in verbatim mode, we copy the entire page 2 into page 1 and return. Otherwise, if we did place a player sprite, reveal the screen. If we didn't place a player sprite, that's an error!

85c ⟨*level draw routine* 77⟩+≡ (278) ◁85b 86▷

```
    LDA     VERBATIM
    BEQ     .copy_page2_to_page1

    LDA     PLAYER_COL
    BPL     .reveal_screen

    SEC                             ; Oops, no player! Return error.
    RTS
```

Uses PLAYER_COL 80c and VERBATIM 81e.

To copy the page, we'll need that second ROW_ADDR2 pointer.

86 ⟨*level draw routine* 77⟩+≡ (278) ◁85c 87▷

```
.copy_page2_to_page1:
    LDA     #$20
    STA     ROW_ADDR2+1
    LDA     #$40
    STA     ROW_ADDR+1
    LDA     #$00
    STA     ROW_ADDR2
    STA     ROW_ADDR
    TAY

.copy_loop:
    LDA     (ROW_ADDR),Y
    STA     (ROW_ADDR2),Y
    INY
    BNE     .copy_loop

    INC     ROW_ADDR2+1
    INC     ROW_ADDR+1
    LDX     ROW_ADDR+1
    CPX     #$60
    BCC     .copy_loop

    CLC
    RTS
```

Uses ROW_ADDR 28b and ROW_ADDR2 28b.

Revealing the screen, using an iris wipe. Then, we remove the guard and player sprites!

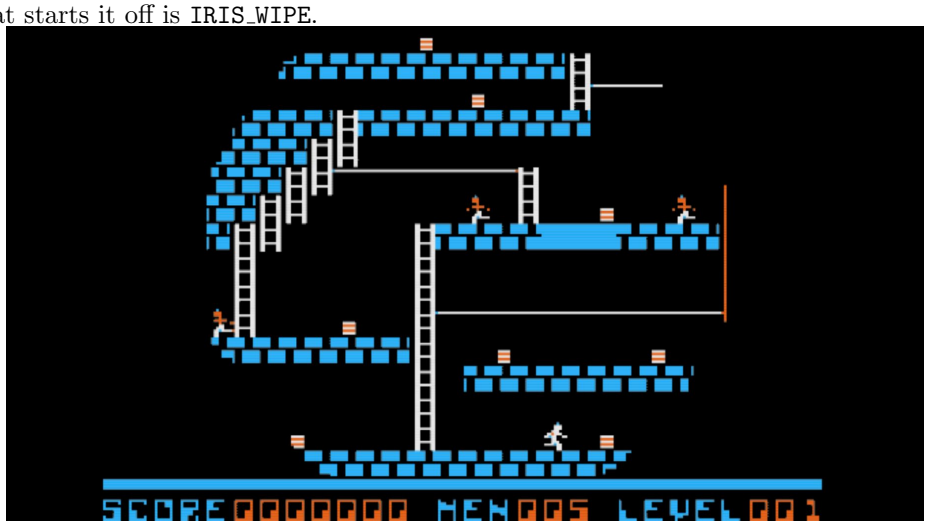
87 ⟨*level draw routine* 77⟩+≡ (278) ◁86

```
.reveal_screen
    JSR     IRIS_WIPE

    LDY     #MAX_GAME_ROW
    STY     GAME_ROWNUM

.row_loop2:
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     #MAX_GAME_COL
    STY     GAME_COLNUM

.col_loop2:
    LDA     (PTR1),Y
    CMP     #SPRITE_PLAYER
    BEQ     .remove
    CMP     #SPRITE_GUARD
    BNE     .next

.remove:
    LDA     #SPRITE_EMPTY
    JSR     DRAW_SPRITE_PAGE2

.next:
    DEC     GAME_COLNUM
    LDY     GAME_COLNUM
    BPL     .col_loop2

    DEC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    BPL     .row_loop2

    CLC
    RTS
```

Uses DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, GAME_ROWNUM 34a, IRIS_WIPE 89, and PTR1 78b.

6.2 Iris Wipe

Whenever a level is finished or starts, there's an iris wipe transition. The routine that starts it off is IRIS_WIPE.



88 ⟨*defines* 4⟩+≡ (281) ◁84b 91▷

```
WIPE_COUNTER        EQU     $6D
WIPE_MODE           EQU     $A5     ; 0 for open, 1 for close.
WIPE_DIR            EQU     $72     ; 0 for close, 1 for open.
WIPE_CENTER_X       EQU     $74
WIPE_CENTER_Y       EQU     $73
```

Defines:
WIPE_COUNTER, used in chunks 89, 100, and 101.
WIPE_MODE, used in chunks 89 and 246.

89 ⟨*iris wipe* 89⟩≡ (278)

```
        ORG     $88A2
    IRIS_WIPE:
        SUBROUTINE

        LDA     #88
        STA     WIPE_CENTER_Y
        LDA     #140
        STA     WIPE_CENTER_X

        LDA     WIPE_MODE
        BEQ     .iris_open

        LDX     #$AA
        STX     WIPE_COUNTER
        LDX     #$00
        STX     WIPE_DIR            ; Close

    .loop_close:
        JSR     IRIS_WIPE_STEP
        DEC     WIPE_COUNTER
        BNE     .loop_close

    .iris_open:
        LDA     #$01
        STA     WIPE_COUNTER
        STA     WIPE_MODE           ; So next time we will close.
        STA     WIPE_DIR            ; Open
        JSR     PUT_STATUS_LIVES
        JSR     PUT_STATUS_LEVEL

    .loop_open:
        JSR     IRIS_WIPE_STEP
        INC     WIPE_COUNTER
        LDA     WIPE_COUNTER
        CMP     #$AA
        BNE     .loop_open
        RTS
```

Defines:
IRIS_WIPE, used in chunk 87.
Uses IRIS_WIPE_STEP 92, PUT_STATUS_LEVEL 53, PUT_STATUS_LIVES 53, WIPE_COUNTER 88, and WIPE_MODE 88.

The routine IRIS_WIPE_STEP does a lot of math to compute the circular iris, all parameterized on WIPE_COUNTER.

Here is a routine that divides a 16-bit value in A and X (X being LSB) by 7, storing the result in Y, with remainder in A. The routine effectively does long division. It also uses two temporaries.

90 ⟨*routines* 5⟩+≡ (281) ◁35 278▷

```
    ORG     $8A45
DIV_BY_7:
    SUBROUTINE
    ; Enter routine with AX set to (unsigned) numerator.
    ; On exit, Y will contain the integer portion of AX/7,
    ; and A contains the remainder.

    STX     MATH_TMPL
    LDY     #$08
    SEC
    SBC     #$07

.loop:
    PHP
    ROL     MATH_TMPH
    ASL     MATH_TMPL
    ROL
    PLP
    BCC     .adjust_up
    SBC     #$07
    JMP     .next

.adjust_up
    ADC     #$07

.next
    DEY
    BNE     .loop

    BCS     .no_adjust
    ADC     #$07
    CLC

.no_adjust
    ROL     MATH_TMPH
    LDY     MATH_TMPH
    RTS
```

Defines:
DIV_BY_7, used in chunk 101.
Uses MATH_TMPH 4 and MATH_TMPL 4.

Now, for one iris wipe step, we will need lots and lots of temporaries.

91 ⟨*defines* 4⟩+≡ (281) ◁88 106a▷

```
WIPE0       EQU     $69     ; 16-bit value
WIPE1       EQU     $67     ; 16-bit value
WIPE2       EQU     $6B     ; 16-bit value
WIPE3L      EQU     $75
WIPE4L      EQU     $76
WIPE5L      EQU     $77
WIPE6L      EQU     $78
WIPE3H      EQU     $79
WIPE4H      EQU     $7A
WIPE5H      EQU     $7B
WIPE6H      EQU     $7C
WIPE7D      EQU     $7D     ; Dividends
WIPE8D      EQU     $7E
WIPE9D      EQU     $7F
WIPE10D     EQU     $80
WIPE7R      EQU     $81     ; Remainders
WIPE8R      EQU     $82
WIPE9R      EQU     $83
WIPE10R     EQU     $84
```

Defines:
WIPE0, used in chunks 99, 100, 103, and 233.
WIPE1, used in chunks 99b, 100b, and 102–104.
WIPE10D, used in chunks 96, 97, 101d, and 104b.
WIPE10R, used in chunks 96, 97, 101d, and 104b.
WIPE2, used in chunks 93, 100, 102, and 103a.
WIPE3H, used in chunks 95, 100e, and 104a.
WIPE3L, used in chunks 95, 100e, and 104a.
WIPE4H, used in chunks 97, 100f, and 105a.
WIPE4L, used in chunks 97, 100f, and 105a.
WIPE5H, used in chunks 96, 100f, and 105b.
WIPE5L, used in chunks 96, 100f, and 105b.
WIPE6H, used in chunks 94b, 101a, and 104d.
WIPE6L, used in chunks 94b, 101a, and 104d.
WIPE7D, used in chunks 96, 97, 101b, and 104c.
WIPE7R, used in chunks 96, 97, 101b, and 104c.
WIPE8D, used in chunks 94b, 95, 101c, and 105c.
WIPE8R, used in chunks 94b, 95, 101c, and 105c.
WIPE9D, used in chunks 94b, 95, 101c, and 104f.
WIPE9R, used in chunks 94b, 95, 101c, and 104f.

The first thing we do for a single step is initialize all those variables!

92 ⟨*iris wipe step* 92⟩≡ (278) 93▷

```
    ORG     $88D7
IRIS_WIPE_STEP:
    SUBROUTINE

⟨WIPE0 = WIPE_COUNTER 100a⟩
⟨WIPE1 = 0 100b⟩
⟨WIPE2 = 2 * WIPE0 100c⟩
⟨WIPE2 = 3 - WIPE2 100d⟩

; WIPE3, WIPE4, WIPE5, and WIPE6 correspond to
; row numbers. WIPE3 is above the center, WIPE6
; is below the center, while WIPE4 and WIPE5 are on
; the center.

⟨WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER 100e⟩
⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 100f⟩
⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 101a⟩

; WIPE7, WIPE8, WIPE9, and WIPE10 correspond to
; column byte numbers. Note the division by 7 pixels!
; WIPE7 is left of center, WIPE10 is right of center,
; while WIPE8 and WIPE9 are on the center.

⟨WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7 101b⟩
⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 101c⟩
⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 101d⟩
```

Defines:
IRIS_WIPE_STEP, used in chunk 89.

Now we loop. This involves checking WIPE1 against WIPE0:

- If WIPE1 < WIPE0, return.
- If WIPE1 == WIPE0, go to DRAW_WIPE_STEP then return.
- Otherwise, call DRAW_WIPE_STEP and go round the loop.

Going around the loop involves calling DRAW_WIPE_STEP, then adjusting the numbers.

93 ⟨*iris wipe step* 92⟩+≡ (278) ◁92

```
.loop:

⟨iris wipe loop check 99b⟩

    JSR     DRAW_WIPE_STEP

    LDA     WIPE2+1
    BPL     .89a7

⟨WIPE2 += 4 * WIPE1 + 6 102⟩
    JMP     .8a14

.89a7:

⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 103a⟩
⟨Decrement WIPE0 103b⟩
⟨Increment WIPE3 104a⟩
⟨Decrement WIPE10 modulo 7 104b⟩
⟨Increment WIPE7 modulo 7 104c⟩
⟨Decrement WIPE6 104d⟩

.8a14:

⟨Increment WIPE1 104e⟩
⟨Increment WIPE9 modulo 7 104f⟩
⟨Decrement WIPE4 105a⟩
⟨Increment WIPE5 105b⟩
⟨Decrement WIPE8 modulo 7 105c⟩
    JMP     .loop
```

Uses DRAW_WIPE_STEP 94a and WIPE2 91.

Drawing a wipe step draws all four parts. There are two rows which move north and two rows that move south. There are also two left and right offsets, one short and one long. This makes eight combinations.

94a ⟨*draw wipe step* 94a⟩≡ (278)

```
        ORG     $8A69
    DRAW_WIPE_STEP:
        SUBROUTINE

    ⟨Draw wipe for south part 94b⟩
    ⟨Draw wipe for north part 95⟩
    ⟨Draw wipe for north2 part 96⟩
    ⟨Draw wipe for south2 part 97⟩
```

Defines:
DRAW_WIPE_STEP, used in chunks 93 and 99b.

Each part consists of two halves, right and left (or east and west).

94b ⟨*Draw wipe for south part* 94b⟩≡ (94a)

```
        LDY     WIPE6H
        BNE     .draw_north
        LDY     WIPE6L
        CPY     #176
        BCS     .draw_north         ; Skip if WIPE6 >= 176

        JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

        ; East side
        LDY     WIPE9D
        CPY     #40
        BCS     .draw_south_west
        LDX     WIPE9R
        JSR     DRAW_WIPE_BLOCK

    .draw_south_west
        ; West side
        LDY     WIPE8D
        CPY     #40
        BCS     .draw_north
        LDX     WIPE8R
        JSR     DRAW_WIPE_BLOCK
```

Uses DRAW_WIPE_BLOCK 98, ROW_TO_ADDR_FOR_BOTH_PAGES 29a, WIPE6H 91, WIPE6L 91, WIPE8D 91, WIPE8R 91, WIPE9D 91, and WIPE9R 91.

95 ⟨*Draw wipe for north part* 95⟩≡ (94a)

```
.draw_north:
    LDY     WIPE3H
    BNE     .draw_north2
    LDY     WIPE3L
    CPY     #176
    BCS     .draw_north2        ; Skip if WIPE3 >= 176

    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

    ; East side
    LDY     WIPE9D
    CPY     #40
    BCS     .draw_north_west
    LDX     WIPE9R
    JSR     DRAW_WIPE_BLOCK

.draw_north_west
    ; West side
    LDY     WIPE8D
    CPY     #40
    BCS     .draw_north2
    LDX     WIPE8R
    JSR     DRAW_WIPE_BLOCK
```

Uses DRAW_WIPE_BLOCK 98, ROW_TO_ADDR_FOR_BOTH_PAGES 29a, WIPE3H 91, WIPE3L 91, WIPE8D 91, WIPE8R 91, WIPE9D 91, and WIPE9R 91.

96 ⟨*Draw wipe for north2 part* 96⟩≡ (94a)

```
.draw_north2:
    LDY     WIPE5H
    BNE     .draw_south2
    LDY     WIPE5L
    CPY     #176
    BCS     .draw_south2        ; Skip if WIPE5 >= 176

    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

    ; East side
    LDY     WIPE10D
    CPY     #40
    BCS     .draw_north2_west
    LDX     WIPE10R
    JSR     DRAW_WIPE_BLOCK

.draw_north2_west
    ; West side
    LDY     WIPE7D
    CPY     #40
    BCS     .draw_south2
    LDX     WIPE7R
    JSR     DRAW_WIPE_BLOCK
```

Uses DRAW_WIPE_BLOCK 98, ROW_TO_ADDR_FOR_BOTH_PAGES 29a, WIPE10D 91, WIPE10R 91, WIPE5H 91, WIPE5L 91, WIPE7D 91, and WIPE7R 91.

97 ⟨*Draw wipe for south2 part* 97⟩≡ (94a)

```
.draw_south2:
    LDY     WIPE4H
    BNE     .end
    LDY     WIPE4L
    CPY     #176
    BCS     .end        ; Skip if WIPE4 >= 176

    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

    ; East side
    LDY     WIPE10D
    CPY     #40
    BCS     .draw_south2_west
    LDX     WIPE10R
    JSR     DRAW_WIPE_BLOCK

.draw_south2_west
    ; West side
    LDY     WIPE7D
    CPY     #40
    BCS     .end
    LDX     WIPE7R
    JMP     DRAW_WIPE_BLOCK           ; tail call

.end:
    RTS
```

Uses DRAW_WIPE_BLOCK 98, ROW_TO_ADDR_FOR_BOTH_PAGES 29a, WIPE10D 91, WIPE10R 91, WIPE4H 91, WIPE4L 91, WIPE7D 91, and WIPE7R 91.

Drawing a wipe block depends on whether we're opening or closing on the level. Closing on the level just blacks out pixels on page 1. Opening on the level copies some pixels from page 2 into page 1.

98 ⟨*draw wipe block* 98⟩≡ (278)

```
      ORG     $8AF6
  DRAW_WIPE_BLOCK:
      SUBROUTINE
      ; Enter routine with X set to the column byte and Y set to
      ; the pixel number within that byte (0-6). ROW_ADDR and
      ; ROW_ADDR2 must contain the base row address for page 1
      ; and page 2, respectively.

      LDA     WIPE_DIR
      BNE     .open

      LDA     (ROW_ADDR),Y
      AND     WIPE_BLOCK_CLOSE_MASK,X
      STA     (ROW_ADDR),Y
      RTS

  .open:
      LDA     (ROW_ADDR2),Y
      AND     WIPE_BLOCK_OPEN_MASK,X
      ORA     (ROW_ADDR),Y
      STA     (ROW_ADDR),Y
      RTS
```

Defines:
DRAW_WIPE_BLOCK, used in chunks 94–97.
Uses ROW_ADDR 28b, ROW_ADDR2 28b, WIPE_BLOCK_CLOSE_MASK 99a, and WIPE_BLOCK_OPEN_MASK 99a.

99a ⟨*tables* 9⟩+≡ (281) ◁82a 110▷

```
      ORG     $8B0C
  WIPE_BLOCK_CLOSE_MASK:
      BYTE     %11110000
      BYTE     %11110000
      BYTE     %11110000
      BYTE     %11110000
      BYTE     %10001111
      BYTE     %10001111
      BYTE     %10001111
  WIPE_BLOCK_OPEN_MASK:
      BYTE     %10001111
      BYTE     %10001111
      BYTE     %10001111
      BYTE     %10001111
      BYTE     %11110000
      BYTE     %11110000
      BYTE     %11110000
```

Defines:
WIPE_BLOCK_CLOSE_MASK, used in chunk 98.
WIPE_BLOCK_OPEN_MASK, used in chunk 98.

99b ⟨*iris wipe loop check* 99b⟩≡ (93)

```
      LDA     WIPE1+1
      CMP     WIPE0+1
      BCC     .draw_wipe_step ; Effectively, if WIPE1 > WIPE0, jump to .draw_wipe_step.
      BEQ     .8969           ; Otherwise jump to .loop1, which...

  .loop1:
      LDA     WIPE1
      CMP     WIPE0
      BNE     .end
      LDA     WIPE1+1
      CMP     WIPE0+1
      BNE     .end            ; If WIPE0 != WIPE1, return.
      JMP     DRAW_WIPE_STEP

  .end:
      RTS

  .8969:
      LDA     WIPE1
      CMP     WIPE0
      BCS     .loop1          ; The other half of the comparison from .loop.

  .draw_wipe_step:
```

Uses DRAW_WIPE_STEP 94a, WIPE0 91, and WIPE1 91.

6.2.1 Initialization

100a ⟨WIPE0 = WIPE_COUNTER 100a⟩≡ (92)

```
        LDA     WIPE_COUNTER
        STA     WIPE0
        LDA     #$00
        STA     WIPE0+1         ; WIPE0 = WIPE_COUNTER
```

Uses WIPE0 91 and WIPE_COUNTER 88.

100b ⟨WIPE1 = 0 100b⟩≡ (92)

```
        ; fallthrough with A = 0
        STA     WIPE1
        STA     WIPE1+1         ; WIPE1 = 0
```

Uses WIPE1 91.

100c ⟨WIPE2 = 2 * WIPE0 100c⟩≡ (92)

```
        LDA     WIPE0
        ASL
        STA     WIPE2
        LDA     WIPE0+1
        ROL
        STA     WIPE2+1         ; WIPE2 = 2 * WIPE0
```

Uses WIPE0 91 and WIPE2 91.

100d ⟨WIPE2 = 3 - WIPE2 100d⟩≡ (92)

```
        LDA     #$03
        SEC
        SBC     WIPE2
        STA     WIPE2
        LDA     #$00
        SBC     WIPE2+1
        STA     WIPE2+1         ; WIPE2 = 3 - WIPE2
```

Uses WIPE2 91.

100e ⟨WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER 100e⟩≡ (92)

```
        LDA     WIPE_CENTER_Y
        SEC
        SBC     WIPE_COUNTER
        STA     WIPE3L
        LDA     #$00
        SBC     #$00
        STA     WIPE3H          ; WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER
```

Uses WIPE3H 91, WIPE3L 91, and WIPE_COUNTER 88.

100f ⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 100f⟩≡ (92)

```
        LDA     WIPE_CENTER_Y
        STA     WIPE4L
        STA     WIPE5L
        LDA     #$00
        STA     WIPE4H
        STA     WIPE5H          ; WIPE4 = WIPE5 = WIPE_CENTER_Y
```

Uses WIPE4H 91, WIPE4L 91, WIPE5H 91, and WIPE5L 91.

101a ⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 101a⟩≡ (92)

```
LDA     WIPE_CENTER_Y
CLC
ADC     WIPE_COUNTER
STA     WIPE6L
LDA     #$00
ADC     #$00
STA     WIPE6H          ; WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER
```

Uses WIPE6H 91, WIPE6L 91, and WIPE_COUNTER 88.

101b ⟨WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7 101b⟩≡ (92)

```
LDA     WIPE_CENTER_X
SEC
SBC     WIPE_COUNTER
TAX
LDA     #$00
SBC     #$00
JSR     DIV_BY_7
STY     WIPE7D
STA     WIPE7R          ; WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7
```

Uses DIV_BY_7 90, WIPE7D 91, WIPE7R 91, and WIPE_COUNTER 88.

101c ⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 101c⟩≡ (92)

```
LDX     WIPE_CENTER_X
LDA     #$00
JSR     DIV_BY_7
STY     WIPE8D
STY     WIPE9D
STA     WIPE8R
STA     WIPE9R          ; WIPE8 = WIPE9 = WIPE_CENTER_X / 7
```

Uses DIV_BY_7 90, WIPE8D 91, WIPE8R 91, WIPE9D 91, and WIPE9R 91.

101d ⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 101d⟩≡ (92)

```
LDA     WIPE_CENTER_X
CLC
ADC     WIPE_COUNTER
TAX
LDA     #$00
ADC     #$00
JSR     DIV_BY_7
STY     WIPE10D
STA     WIPE10R         ; WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7
```

Uses DIV_BY_7 90, WIPE10D 91, WIPE10R 91, and WIPE_COUNTER 88.

6.2.2 All that math stuff

102 ⟨WIPE2 += 4 * WIPE1 + 6 102⟩≡ (93)

```
      LDA     WIPE1
      ASL
      STA     MATH_TMPL
      LDA     WIPE1+1
      ROL
      STA     MATH_TMPH       ; MATH_TMP = WIPE1 * 2

      LDA     MATH_TMPL
      ASL
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ROL
      STA     MATH_TMPH       ; MATH_TMP *= 2

      LDA     WIPE2
      CLC
      ADC     MATH_TMPL
      STA     MATH_TMPL
      LDA     WIPE2+1
      ADC     MATH_TMPH
      STA     MATH_TMPH       ; MATH_TMP += WIPE2

      LDA     #$06
      CLC
      ADC     MATH_TMPL
      STA     WIPE2
      LDA     #$00
      ADC     MATH_TMPH
      STA     WIPE2+1        ; WIPE2 = MATH_TMP + 6
```

Uses MATH_TMPH 4, MATH_TMPL 4, WIPE1 91, and WIPE2 91.

103a ⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 103a⟩≡ (93)

```
      LDA     WIPE1
      SEC
      SBC     WIPE0
      STA     MATH_TMPL
      LDA     WIPE1+1
      SBC     WIPE0+1
      STA     MATH_TMPH       ; MATH_TMP = WIPE1 - WIPE0

      LDA     MATH_TMPL
      ASL
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ROL
      STA     MATH_TMPH       ; MATH_TMP *= 2

      LDA     MATH_TMPL
      ASL
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ROL
      STA     MATH_TMPH       ; MATH_TMP *= 2

      LDA     MATH_TMPL
      CLC
      ADC     #$10
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ADC     #$00
      STA     MATH_TMPH       ; MATH_TMP += 16

      LDA     MATH_TMPL
      CLC
      ADC     WIPE2
      STA     WIPE2
      LDA     MATH_TMPH
      ADC     WIPE2+1
      STA     WIPE2+1        ; WIPE2 += MATH_TMP
```

Uses MATH_TMPH 4, MATH_TMPL 4, WIPE0 91, WIPE1 91, and WIPE2 91.

103b ⟨*Decrement* WIPE0 103b⟩≡ (93)

```
      LDA     WIPE0
      PHP
      DEC     WIPE0
      PLP
      BNE     .b9ec
      DEC     WIPE0+1         ; WIPE0--
  .b9ec
```

Uses WIPE0 91.

104a ⟨*Increment* WIPE3 104a⟩≡ (93)

```
        INC     WIPE3L
        BNE     .89f2
        INC     WIPE3H          ; WIPE3++
    .89f2
```

Uses WIPE3H 91 and WIPE3L 91.

104b ⟨*Decrement* WIPE10 *modulo 7* 104b⟩≡ (93)

```
        DEC     WIPE10R
        BPL     .89fc
        LDA     #$06
        STA     WIPE10R
        DEC     WIPE10D
    .89fc
```

Uses WIPE10D 91 and WIPE10R 91.

104c ⟨*Increment* WIPE7 *modulo 7* 104c⟩≡ (93)

```
        INC     WIPE7R
        LDA     WIPE7R
        CMP     #$07
        BNE     .8a0a
        LDA     #$00
        STA     WIPE7R
        INC     WIPE7D
    .8a0a
```

Uses WIPE7D 91 and WIPE7R 91.

104d ⟨*Decrement* WIPE6 104d⟩≡ (93)

```
        DEC     WIPE6L
        LDA     WIPE6L
        CMP     #$FF
        BNE     .8a14
        DEC     WIPE6H
```

Uses WIPE6H 91 and WIPE6L 91.

104e ⟨*Increment* WIPE1 104e⟩≡ (93)

```
        INC     WIPE1
        BNE     .8a1a
        INC     WIPE1+1         ; WIPE1++
    .8a1a
```

Uses WIPE1 91.

104f ⟨*Increment* WIPE9 *modulo 7* 104f⟩≡ (93)

```
        INC     WIPE9R
        LDA     WIPE9R
        CMP     #$07
        BNE     .8a28
        LDA     #$00
        STA     WIPE9R
        INC     WIPE9D
    .8a28
```

Uses WIPE9D 91 and WIPE9R 91.

105a ⟨*Decrement* WIPE4 105a⟩≡ (93)

```
        DEC     WIPE4L
        LDA     WIPE4L
        CMP     #$FF
        BNE     .8a32
        DEC     WIPE4H
    .8a32
```

Uses WIPE4H 91 and WIPE4L 91.

105b ⟨*Increment* WIPE5 105b⟩≡ (93)

```
        INC     WIPE5L
        BNE     .8a38
        INC     WIPE5H          ; WIPE5++
    .8a38
```

Uses WIPE5H 91 and WIPE5L 91.

105c ⟨*Decrement* WIPE8 *modulo 7* 105c⟩≡ (93)

```
        DEC     WIPE8R
        BPL     .8a42
        LDA     #$06
        STA     WIPE8R
        DEC     WIPE8D
    .8a42
```

Uses WIPE8D 91 and WIPE8R 91.

6.3 Level data

Now that we have the ability to draw a level from level data, we need a routine to get that level data. Recall that level data needs to be stored in pointers specified in the CURR_LEVEL_ROW_SPRITES_PTR_ tables.

6.3.1 Getting the compressed level data

The level data is stored in the game in compressed form, so we first grab the data for the level and put it into the 256-byte DISK_BUFFER buffer. This buffer is the same as the DOS read/write buffer, so that level data can be loaded directly from disk. Levels on disk are stored starting at track 3 sector 0, with levels being stored in consecutive sectors, 16 per track.

There's one switch here, GAME_MODE, which dictates whether we're going to display the high-score screen, attract-mode game play, the splash screen, or an actual level for playing.

Also, if we're in attract mode, instead of loading the level from disk, we load the level from the game image, which contains the first three "standard" levels.

One additional feature is that you can start the routine with A being 1 to read a level, 2 to write a level, and 4 to format the entire disk. Writing and formatting is used by the level editor. But see also FORMAT_PATCH for prevention of formatting Lode Runner itself.

106a ⟨*defines* 4⟩+≡ (281) ◁91 107▷

```
GAME_MODE               EQU     $A7

GAME_MODE_SPLASH_SCREEN         EQU     #$00
GAME_MODE_ATTRACT_MODE          EQU     #$01
GAME_MODE_PLAY_MODE             EQU     #$02
GAME_MODE_PLAY_IN_EDITOR        EQU     #$03
GAME_MODE_4                     EQU     #$04
GAME_MODE_LEVEL_EDITOR          EQU     #$05

DISK_BUFFER             EQU     $0D00       ; 256 bytes
RWTS_ADDR               EQU     $24         ; 2 bytes
DISK_LEVEL_LOC          EQU     $96
```

Defines:
GAME_MODE, used in chunks 108, 124, 132–34, 139, 246, 250, 259, and 263.

106b ⟨*jump to RWTS indirectly* 106b⟩≡ (278)

```
JMP_RWTS        EQU     $23     ; JMP $0000, gets loaded with RWTS address later
```

Defines:
JMP_RWTS, used in chunk 108.

107 ⟨*defines* 4⟩+≡ (281) ◁106a 111a▷

```
DISK_ACCESS_READ        EQU     #$01
DISK_ACCESS_WRITE       EQU     #$02
DISK_ACCESS_FORMAT      EQU     #$04
```

Defines:
DISK_ACCESS_FORMAT, never used.
DISK_ACCESS_READ, used in chunk 111b.
DISK_ACCESS_WRITE, never used.

108 ⟨*load compressed level data* 108⟩≡ (278)

```
    ORG     $630E
ACCESS_COMPRESSED_LEVEL_DATA:
    SUBROUTINE
    ; Enter routine with A set to command: 1 = read, 2 = write, 4 = format

    STA     IOB_COMMAND_CODE
    LDA     GAME_MODE
    LSR
    ; If GAME_MODE is 0 or 1, copy level data from image
    BEQ     .copy_level_data_from_image

    ; Otherwise, read/write/format level on disk
    LDA     DISK_LEVEL_LOC
    LSR
    LSR
    LSR
    LSR
    CLC
    ADC     #$03
    STA     IOB_TRACK_NUMBER            ; track 3 + (DISK_LEVEL_LOC >> 4)
    LDA     DISK_LEVEL_LOC
    AND     #$0F
    STA     IOB_SECTOR_NUMBER           ; sector DISK_LEVEL_LOC & 0x0F
    LDA     #<DISK_BUFFER
    STA     IOB_READ_WRITE_BUFFER_PTR
    LDA     #>DISK_BUFFER
    STA     IOB_READ_WRITE_BUFFER_PTR+1 ; IOB_READ_WRITE_BUFFER_PTR = 0D00
    LDA     #$00
    STA     IOB_VOLUME_NUMBER_EXPECTED  ; any volume

ACCESS_DISK_OR_RESET_GAME:
    LDY     #<DOS_IOB
    LDA     #>DOS_IOB
    JSR     JMP_RWTS
    BCC     .end
    JMP     RESET_GAME      ; On error

.end:
    RTS

.copy_level_data_from_image:
    ⟨Copy level data 109a⟩
```

Uses DOS_IOB 229, GAME_MODE 106a, IOB_COMMAND_CODE 229, IOB_READ_WRITE_BUFFER_PTR 229, IOB_SECTOR_NUMBER 229, IOB_TRACK_NUMBER 229, IOB_VOLUME_NUMBER_EXPECTED 229, and JMP_RWTS 106b.

We're not really using ROW_ADDR here as a row address, just as a convenient place to store a pointer. Also, we can see that the attract-mode level data is stored in 256-byte pages at 9F00, A000, and A100. Level numbers start from 1, so 9E00 doesn't actually contain level data.

Since the game is supposed to come with 150 levels, there is not enough room to store all of it, so the rest of the level data must be on disk. Only the first few levels are in memory.

109a ⟨*Copy level data* 109a⟩≡ (108)

```
⟨ROW_ADDR = $9E00 + LEVELNUM * $0100 109b⟩
⟨Copy data from ROW_ADDR into DISK_BUFFER 109c⟩
```

109b ⟨ROW_ADDR = $9E00 + LEVELNUM * $0100 109b⟩≡ (109a)

```
    LDA     LEVELNUM        ; 1-based
    CLC
    ADC     #$9E
    STA     ROW_ADDR+1
    LDY     #$00
    STY     ROW_ADDR        ; ROW_ADDR <- 9E00 + LEVELNUM * 0x100
```

Uses LEVELNUM 52 and ROW_ADDR 28b.

109c ⟨*Copy data from* ROW_ADDR *into* DISK_BUFFER 109c⟩≡ (109a)

```
.copyloop:
    LDA     (ROW_ADDR),Y
    STA     DISK_BUFFER,Y
    INY
    BNE     .copyloop
    RTS
```

Uses ROW_ADDR 28b.

Since levels are 28 sprites across with two sprites per byte, we'll show the hex data as 14 bytes per line. There is no level data past the 16th line.

110 ⟨*tables* 9⟩+≡ (281) ◁99a 119a▷

```
    ORG     $9F00
LEVEL1_DATA:
    HEX     06 00 00 07 00 00 07 00 00 06 00 00 03 06
    HEX     13 11 11 11 11 11 11 11 11 03 00 00 03 06
    HEX     03 00 00 00 00 00 00 07 00 43 44 44 03 76
    HEX     11 11 11 11 31 11 11 11 11 01 00 00 13 11
    HEX     11 11 11 11 33 00 00 00 00 00 00 00 03 00
    HEX     11 11 11 31 43 44 44 44 03 00 00 00 03 00
    HEX     71 70 11 33 00 00 00 08 03 70 00 08 03 70
    HEX     11 11 31 03 00 00 13 11 21 22 11 11 13 11
    HEX     00 00 30 00 00 00 03 00 00 00 00 00 03 00
    HEX     00 00 30 00 00 00 03 00 00 00 00 00 03 00
    HEX     00 00 38 00 70 00 43 44 44 44 44 44 03 70
    HEX     13 11 11 11 11 11 03 70 00 00 70 00 13 11
    HEX     03 00 00 00 00 00 03 11 11 11 11 01 03 00
    HEX     03 00 00 00 00 00 03 00 00 00 00 00 03 00
    HEX     03 00 00 70 00 00 03 00 90 70 00 00 03 00
    HEX     11 11 11 11 11 11 11 11 11 11 11 11 11 11
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
LEVEL2_DATA:
    HEX     11 11 61 11 11 11 11 11 11 11 11 01 11 11
    HEX     06 00 61 01 44 44 44 04 00 00 00 00 00 10
    HEX     16 61 60 01 71 00 10 13 00 00 00 00 80 10
    HEX     06 11 11 01 11 11 11 13 11 11 07 31 11 11
    HEX     61 00 00 47 44 11 11 13 11 11 11 31 01 11
    HEX     11 11 11 01 00 01 10 13 11 10 11 31 01 11
    HEX     00 00 00 80 00 00 10 13 11 10 01 31 71 10
    HEX     13 11 11 11 13 73 10 13 11 10 01 31 11 11
    HEX     03 00 00 00 13 11 11 13 11 07 01 31 11 10
    HEX     03 00 00 00 00 11 11 13 11 11 01 31 11 17
    HEX     03 00 00 00 01 00 00 03 00 00 00 30 00 10
    HEX     13 11 13 11 31 21 21 21 21 21 21 31 21 21
    HEX     13 11 13 11 31 11 11 11 00 11 11 31 21 21
    HEX     03 00 03 00 30 11 11 11 01 10 11 31 00 00
    HEX     13 11 11 11 31 11 11 11 11 07 11 11 11 31
    HEX     93 00 00 00 30 70 10 11 11 01 80 00 00 30
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
LEVEL3_DATA:
    HEX     00 00 00 30 00 00 00 00 00 00 00 00 00 00
    HEX     80 07 00 30 00 00 00 00 00 00 00 00 70 08
    HEX     13 81 07 30 00 00 00 00 00 00 00 70 18 31
    HEX     03 10 01 30 11 11 36 11 11 31 00 10 01 30
    HEX     03 00 00 00 00 00 33 07 00 30 00 00 00 30
    HEX     03 00 00 00 00 00 37 03 00 30 00 00 00 30
```

```
HEX     03 00 00 00 00 00 33 07 00 00 00 00 00 30
HEX     03 00 00 00 00 00 37 03 00 00 00 00 00 30
HEX     03 00 00 00 00 00 33 07 00 00 00 00 00 30
HEX     03 00 00 00 00 00 37 03 00 00 00 00 00 30
HEX     03 00 00 00 00 00 33 07 00 00 00 00 00 30
HEX     03 00 00 00 00 00 37 03 00 00 00 00 00 30
HEX     03 00 00 00 00 00 33 07 00 00 00 00 00 30
HEX     03 00 00 00 00 00 37 03 00 00 00 00 00 30
HEX     03 00 00 00 09 00 33 07 00 00 00 00 00 30
HEX     03 00 30 11 11 11 11 11 11 11 11 03 00 30
HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Defines:
LEVEL1_DATA, never used.
LEVEL2_DATA, never used.
LEVEL3_DATA, never used.

6.3.2 Uncompressing and displaying the level

Loading the level also sets the player ALIVE flag to 1 (alive). Throughout the code, LSR ALIVE simply sets the flag to 0 (dead).

111a ⟨*defines* 4⟩+≡ (281) ◁107 112a▷

```
ALIVE       EQU     $9A
```

Defines:
ALIVE, used in chunks 43, 111b, 126, 140, 141, 148, 191a, 193, 204, 206, 208, 210, and 250.

111b ⟨*load level* 111b⟩≡ (278)

```
        ORG     $6238
LOAD_LEVEL:
        SUBROUTINE
        ; Enter routine with X set to whether the level should be
        ; loaded verbatim or not.

        STX     VERBATIM

        ⟨Initialize level counts 112b⟩

        LDA     #$01
        STA     ALIVE       ; Set player live
        ; A happens to also be DISK_ACCESS_READ.
        JSR     ACCESS_COMPRESSED_LEVEL_DATA

        ⟨uncompress level data 113⟩
```

Defines:
LOAD_LEVEL, used in chunks 115b, 250, and 265.
Uses ALIVE 111a, DISK_ACCESS_READ 107, and VERBATIM 81e.

112a ⟨*defines* 4⟩+≡ (281) ◁111a 118▷

```
LEVEL_DATA_INDEX        EQU     $92
```

Defines:
LEVEL_DATA_INDEX, used in chunks 112b, 114, and 116.

Here we are initializing variables in preparation for loading the level data. Since drawing the level will keep track of ladder, gold, and guard count, we need to zero them out. There are also some areas of memory whose purpose is not yet known, and these are zeroed out also.

112b ⟨*Initialize level counts* 112b⟩≡ (111b)

```
    LDX     #$FF
    STX     PLAYER_COL
    INX
    STX     LADDER_COUNT
    STX     GOLD_COUNT
    STX     GUARD_COUNT
    STX     GUARD_NUM
    STX     DIG_ANIM_STATE
    STX     LEVEL_DATA_INDEX
    STX     TMP
    STX     GAME_ROWNUM
    TXA

    LDX     #30
.loop1
    STA     BRICK_FILL_TIMERS,X
    DEX
    BPL     .loop1

    LDX     #$05
.loop2
    STA     GUARD_RESURRECTION_TIMERS,X
    DEX
    BPL     .loop2
```

Uses GAME_ROWNUM 34a, GOLD_COUNT 81d, GUARD_COUNT 81d, GUARD_NUM 181, LADDER_COUNT 81d, LEVEL_DATA_INDEX 112a, PLAYER_COL 80c, and TMP 4.

The level data is stored in "compressed" form, just 4 bits per sprite since we don't use any higher ones to define a level. For each of the 16 game rows, we load up the compressed row data and break it apart, one 4-bit sprite per column.

Once we've done that, we draw the level using `DRAW_LEVEL_PAGE2`. That routine returns an error if there was no player sprite in the level. If there was no error, we simply return. Otherwise we have to handle the error condition, since there's no point in playing without a player!

113 ⟨*uncompress level data* 113⟩≡ (111b)

```
    LDY     GAME_ROWNUM
.row_loop:
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    ⟨uncompress row data 114⟩
    ⟨next compressed row for row_loop 115a⟩

    JSR     DRAW_LEVEL_PAGE2
    BCC     .end                ; No error

⟨handle no player sprite in level 115b⟩

.end:
    RTS

.reset_game:
    JMP     RESET_GAME
```

Uses `DRAW_LEVEL_PAGE2` 77 and `GAME_ROWNUM` 34a.

Each row will have their sprite data stored at locations specified by the CURR_LEVEL_ROW_SPRITES_PTR_ tables.

To uncompress the data for a row, we use the counter in TMP as an odd/even switch so that we know which 4-bit chunk (nibble) in a byte we want. Even numbers are for the low nibble while odd numbers are for the high nibble.

In addition, if we encounter any sprite number 10 or above then we replace it with sprite 0 (all black).

114 ⟨*uncompress row data* 114⟩≡ (113)

```
    LDA     #$00
    STA     GAME_COLNUM

.col_loop:
    LDA     TMP                         ; odd/even counter
    LSR
    LDY     LEVEL_DATA_INDEX
    LDA     DISK_BUFFER,Y
    BCS     .628c                       ; odd?
    AND     #$0F
    BPL     .6292                       ; unconditional jump
.628c

    LSR
    LSR
    LSR
    LSR
    INC     LEVEL_DATA_INDEX

.6292
    INC     TMP

    LDY     GAME_COLNUM
    CMP     #10
    BCC     .629c
    LDA     #SPRITE_EMPTY               ; sprite >= 10 -> sprite 0
.629c:

    STA     (PTR1),Y
    STA     (PTR2),Y

    INC     GAME_COLNUM
    LDA     GAME_COLNUM
    CMP     #28
    BCC     .col_loop                   ; loop while GAME_COLNUM < 28
```

Uses GAME_COLNUM 34a, LEVEL_DATA_INDEX 112a, PTR1 78b, PTR2 78b, and TMP 4.

115a ⟨*next compressed row for* `row_loop` 115a⟩≡ (113)

```
INC     GAME_ROWNUM
LDY     GAME_ROWNUM
CPY     #16
BCC     .row_loop                   ; loop while GAME_ROWNUM < 16
```

Uses `GAME_ROWNUM` 34a.

When there's no player sprite in the level, a few things can happen. Firstly, if `DISK_LEVEL_LOC` is zero, we're going to jump to `RESET_GAME`. Otherwise, we set `DISK_LEVEL_LOC` to zero, increment `\$97`, set X to `0xFF`, and retry `LOAD_LEVEL` from the very beginning.

115b ⟨*handle no player sprite in level* 115b⟩≡ (113)

```
LDA     DISK_LEVEL_LOC
BEQ     .reset_game

LDX     #$00
STX     DISK_LEVEL_LOC
INC     GUARD_PATTERN_OFFSET
DEX
JMP     LOAD_LEVEL
```

Uses `GUARD_PATTERN_OFFSET` 245c and `LOAD_LEVEL` 111b.

116 ⟨*dead code* 116⟩≡ (281) 141a▷

```
    ORG     $62C7
COMPRESS_AND_SAVE_LEVEL_DATA:
    SUBROUTINE

    LDA     #$00
    STA     LEVEL_DATA_INDEX
    STA     TMP
    STA     GAME_ROWNUM

.loop:
    LDY     GAME_ROWNUM
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     #$00
    STY     GAME_COLNUM

.loop2:
    LDA     TMP
    LSR
    LDA     (PTR1),Y
    BCS     .shift_left

    STA     SPRITE_NUM
    BPL     .next

.shift_left:
    ASL
    ASL
    ASL
    ASL
    ORA     SPRITE_NUM
    LDY     LEVEL_DATA_INDEX
    STA     DISK_BUFFER,Y
    INC     LEVEL_DATA_INDEX

.next:
    INC     TMP
    INC     GAME_COLNUM
    LDY     GAME_COLNUM
    CPY     #MAX_GAME_COL+1
    BCC     .loop2

    INC     GAME_ROWNUM
    LDA     GAME_ROWNUM
    CMP     #MAX_GAME_ROW+1
    BCC     .loop

    LDA     #DISK_ACCESS_WRITE
    JMP     ACCESS_COMPRESSED_LEVEL_DATA    ; tailcall
```

Defines:

COMPRESS_AND_SAVE_LEVEL_DATA, used in chunk 268.
Uses GAME_COLNUM 34a, GAME_ROWNUM 34a, LEVEL_DATA_INDEX 112a, PTR1 78b, SPRITE_NUM 25c, and TMP 4.

Chapter 7

High scores

For this routine, we have two indexes. The first is stored in `HI_SCORE_INDEX` and is the high score number, from 1 to 10. The second is stored in `HI_SCORE_OFFSET` and keeps our place in the actual high score data table stored at `HI_SCORE_OFFSET`.

There are ten slots in the high score table, each with eight bytes. The first three bytes are for the player initials, the fourth byte is the level – or zero if the row should be empty – and the last four bytes are the BCD-encoded score, most significant byte first.

118 ⟨*defines* 4⟩+≡ (281) ◁112a 122c▷

```
HI_SCORE_INDEX      EQU     $55     ; aliased with TMP_GUARD_COL
HI_SCORE_OFFSET     EQU     $56     ; aliased with TMP_GUARD_ROW
```

Defines:
`HI_SCORE_INDEX`, used in chunks 120–22.
`HI_SCORE_OFFSET`, used in chunks 121 and 122a.

119a ⟨*tables* 9⟩+≡ (281) ◁110 121a▷

```
      ORG     $1F00
  HI_SCORE_DATA:
      HEX     D2 CC D0 06 00 03 10 75 A0 A0 A0 05 00 02 81 25
      HEX     A0 A0 A0 02 00 01 74 25 A0 A0 A0 01 00 00 54 25
      HEX     A0 A0 A0 01 00 00 19 75 A0 A0 A0 01 00 00 15 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 CC CF C4 C5 A0 D2 D5 CE CE C5 D2 FF
```

Defines:
HI_SCORE_DATA, used in chunks 121, 122a, 231, 233, and 244.

119b ⟨*construct and display high score screen* 119b⟩≡ (278)

```
      ORG     $786B

  HI_SCORE_SCREEN:
      SUBROUTINE

      JSR     CLEAR_HGR2
      LDA     #$40
      STA     DRAW_PAGE
      LDA     #$00
      STA     GAME_COLNUM
      STA     GAME_ROWNUM

      ⟨draw high score table header 120a⟩
      ⟨draw high score rows 120b⟩
      ⟨show high score page 123⟩
```

Defines:
HI_SCORE_SCREEN, used in chunks 134c, 145, and 233.
Uses CLEAR_HGR2 5, DRAW_PAGE 45, GAME_COLNUM 34a, and GAME_ROWNUM 34a.

120a ⟨*draw high score table header* 120a⟩≡ (119b)

```
    ; "    LODE RUNNER HIGH SCORES\r"
    ; "\r"
    ; "\r"
    ; "    INITIALS LEVEL  SCORE\r"
    ; "    -------- ----- --------\r"
    JSR     PUT_STRING
    HEX     A0 A0 A0 A0 CC CF C4 C5 A0 D2 D5 CE CE C5 D2 A0
    HEX     C8 C9 C7 C8 A0 D3 C3 CF D2 C5 D3 8D 8D 8D A0 A0
    HEX     A0 A0 C9 CE C9 D4 C9 C1 CC D3 A0 CC C5 D6 C5 CC
    HEX     A0 A0 D3 C3 CF D2 C5 8D A0 A0 A0 A0 AD AD AD AD
    HEX     AD AD AD AD A0 AD AD AD AD AD A0 AD AD AD AD AD
    HEX     AD AD AD 8D 00
```

Uses PUT_STRING 47 and SCORE 50b.

120b ⟨*draw high score rows* 120b⟩≡ (119b)

```
    LDA     #$01
    STA     HI_SCORE_INDEX              ; Used for row number
.loop:
    ⟨draw high score row number 120c⟩
    ⟨draw high score initials 121b⟩
    ⟨draw high score level 121c⟩
    ⟨draw high score 122a⟩
    ⟨next high score row 122b⟩
```

Uses HI_SCORE_INDEX 118.

120c ⟨*draw high score row number* 120c⟩≡ (120b)

```
    CMP     #$0A
    BNE     .display_0_to_9
    LDA     #1
    JSR     PUT_DIGIT
    LDA     #0
    JSR     PUT_DIGIT
    JMP     .rest_of_row_number

.display_0_to_9:
    LDA     #$A0
    JSR     PUT_CHAR        ; space
    LDA     HI_SCORE_INDEX
    JSR     PUT_DIGIT

.rest_of_row_number:
    ; ".    "
    JSR     PUT_STRING
    HEX     AE A0 A0 A0 A0 00
```

Uses HI_SCORE_INDEX 118, PUT_CHAR 46a, PUT_DIGIT 48a, and PUT_STRING 47.

121a ⟨*tables* 9⟩+≡ (281) ◁119a 135a▷

```
    ORG     $79A2
    ; Because table indices are 1-based, there's an extra 00
    ; at the beginning that never gets used.
HI_SCORE_TABLE_OFFSETS:
    HEX     00 00 08 10 18 20 28 30 38 40 48
```

Defines:
HI_SCORE_TABLE_OFFSETS, used in chunks 121b and 233.

121b ⟨*draw high score initials* 121b⟩≡ (120b)

```
    LDX     HI_SCORE_INDEX
    LDY     HI_SCORE_TABLE_OFFSETS,X
    STY     HI_SCORE_OFFSET
    LDA     HI_SCORE_DATA+3,Y
    BNE     .draw_initials
    JMP     .next_high_score_row
.draw_initials:
    LDY     HI_SCORE_OFFSET
    LDA     HI_SCORE_DATA,Y
    JSR     PUT_CHAR
    LDY     HI_SCORE_OFFSET
    LDA     HI_SCORE_DATA+1,Y
    JSR     PUT_CHAR
    LDY     HI_SCORE_OFFSET
    LDA     HI_SCORE_DATA+2,Y
    JSR     PUT_CHAR

    ; "    "
    JSR     PUT_STRING
    HEX     A0 A0 A0 A0 00
```

Uses HI_SCORE_DATA 119a, HI_SCORE_INDEX 118, HI_SCORE_OFFSET 118, HI_SCORE_TABLE_OFFSETS 121a, PUT_CHAR 46a, and PUT_STRING 47.

121c ⟨*draw high score level* 121c⟩≡ (120b)

```
    LDY     HI_SCORE_OFFSET
    LDA     HI_SCORE_DATA+3,Y
    JSR     TO_DECIMAL3
    LDA     HUNDREDS
    JSR     PUT_DIGIT
    LDA     TENS
    JSR     PUT_DIGIT
    LDA     UNITS
    JSR     PUT_DIGIT

    ; "  "
    JSR     PUT_STRING
    HEX     A0 A0 00
```

Uses HI_SCORE_DATA 119a, HI_SCORE_OFFSET 118, HUNDREDS 48b, PUT_DIGIT 48a, PUT_STRING 47, TENS 48b, TO_DECIMAL3 49, and UNITS 48b.

122a ⟨*draw high score* 122a⟩≡ (120b)

```
        LDY     HI_SCORE_OFFSET
        LDA     HI_SCORE_DATA+4,Y
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT

        LDY     HI_SCORE_OFFSET
        LDA     HI_SCORE_DATA+5,Y
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT

        LDY     HI_SCORE_OFFSET
        LDA     HI_SCORE_DATA+6,Y
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT

        LDY     HI_SCORE_OFFSET
        LDA     HI_SCORE_DATA+7,Y
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT
```

Uses BCD_TO_DECIMAL2 50a, HI_SCORE_DATA 119a, HI_SCORE_OFFSET 118, PUT_DIGIT 48a, TENS 48b, and UNITS 48b.

122b ⟨*next high score row* 122b⟩≡ (120b)

```
  .next_high_score_row:
        JSR     NEWLINE
        INC     HI_SCORE_INDEX
        LDA     HI_SCORE_INDEX
        CMP     #11
        BCS     .end
        JMP     .loop
```

Uses HI_SCORE_INDEX 118 and NEWLINE 46a.

122c ⟨*defines* 4⟩+≡ (281) ◁118 130a▷

```
  TXTPAGE2                EQU     $C055
```

Defines:
TXTPAGE2, used in chunks 73a, 123, and 268.

123 ⟨*show high score page* 123⟩≡ (119b)

```
.end:
    STA     TXTPAGE2        ; Flip to page 2
    LDA     #$20
    STA     DRAW_PAGE       ; Set draw page to 1
    RTS
```

Uses DRAW_PAGE 45 and TXTPAGE2 122c.

Chapter 8

Game play

8.1 Splash screen

124 ⟨*splash screen* 124⟩≡ (278)

```
      ORG     $6008
  RESET_GAME:
      SUBROUTINE

      JSR     CLEAR_HGR1

      LDA     #$FF
      STA     .rd_table+1
      LDA     #$0E
      STA     .rd_table+2     ; RD_TABLE = 0x0EFF
      LDY     #$00
      STY     GAME_ROWNUM
      STY     GAME_MODE
      STY     DISK_LEVEL_LOC  ; GAME_ROWNUM = DISK_LEVEL_LOC = GAME_MODE = 0
      LDA     #$20
      STA     HGR_PAGE
      STA     DRAW_PAGE       ; HGR_PAGE = DRAW_PAGE = 0x20

      ⟨splash screen loop 125⟩

      STA     TXTPAGE1
      STA     HIRES
      STA     MIXCLR
      STA     TXTCLR
      JMP     LONG_DELAY
```

Uses CLEAR_HGR1 5, DRAW_PAGE 45, GAME_MODE 106a, GAME_ROWNUM 34a, HGR_PAGE 28b, HIRES 130a, MIXCLR 130a, TXTCLR 130a, and TXTPAGE1 130a.

This loop writes a screen of graphics by reading from the table starting at `$0F00`. The table is in pairs of bytes, where the first byte is the byte offset from the beginning of the row, and the second byte is the byte to write. However, if the first byte is `0x00` then we end that row.

As in other cases, the pointer into the table is stored in the `LDA` instruction that reads from the table.

The code takes advantage of the fact that all bytes written to the page have their high bit set, while offsets from the beginning of the row are always less than `0x80`. Thus, if we read a byte and it is `0x00`, we end the loop. Otherwise, if the byte is less than `0x80` we set that as the offset. Otherwise, the byte has its high bit set, and we write that byte to the graphics page.

125 ⟨*splash screen loop* 125⟩≡ (124)

```
.draw_splash_screen_row:
    JSR     ROW_TO_ADDR     ; ROW_ADDR = ROW_TO_ADDR(Y)
    LDY     #$00

.loop:
    INC     .rd_table+1
    BNE     .rd_table
    INC     .rd_table+2     ; RD_TABLE++

.rd_table:
    LDA     $1A84           ; A <- *RD_TABLE ($1A84 is just a dummy value)
    BEQ     .end_of_row     ; if A == 0: break
    BPL     .is_row_offset  ; if A > 0: A -> Y, .loop
    STA     (ROW_ADDR),Y    ; *(ROW_ADDR+Y) = A

    INY                     ; Y++
    BPL     .loop           ; While Y < 0x80 (really while not 00)

.is_row_offset:
    TAY
    BPL     .loop           ; Unconditional jump

.end_of_row:
    INC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    CPY     #192
    BCC     .draw_splash_screen_row
```

Uses `GAME_ROWNUM` 34a, `ROW_ADDR` 28b, and `ROW_TO_ADDR` 28c.

126 ⟨*handle timers* 126⟩≡ (278)

```
      ORG     $75F4
  HANDLE_TIMERS:
      SUBROUTINE

      JSR     GUARD_RESURRECTIONS

      ; Increment GUARD_RESURRECT_COL mod 29

      INC     GUARD_RESURRECT_COL
      LDA     GUARD_RESURRECT_COL
      CMP     #MAX_GAME_COL+1
      BCC     .guard_col_incremented

      LDA     #$00
      STA     GUARD_RESURRECT_COL

  .guard_col_incremented:
      LDX     #$1E        ; 30

  .loop:
      LDA     BRICK_FILL_TIMERS,X
      STX     TMP_LOOP_CTR
      BNE     .table_ce0_nonzero
      JMP     .next

  .table_ce0_nonzero:
      DEC     BRICK_FILL_TIMERS,X
      BEQ     .brick_fill_timer_expired

      LDA     BRICK_DIG_COLS,X
      STA     GAME_COLNUM
      LDA     BRICK_DIG_ROWS,X
      STA     GAME_ROWNUM

      LDA     BRICK_FILL_TIMERS,X
      CMP     #$14            ; 20
      BNE     .check_for_10

      LDA     #SPRITE_BRICK_FILL0

  .draw_sprite:
      JSR     DRAW_SPRITE_PAGE2
      LDX     GAME_COLNUM
      LDY     GAME_ROWNUM
      JSR     GET_SCREEN_COORDS_FOR
      LDA     #SPRITE_EMPTY
      JSR     ERASE_SPRITE_AT_PIXEL_COORDS

  .next_:
```

```
    JMP     .next

.check_for_10:
    CMP     #$0A            ; 10
    BNE     .next_

    LDA     #SPRITE_BRICK_FILL1
    BNE     .draw_sprite            ; Unconditional

.brick_fill_timer_expired:
    LDX     TMP_LOOP_CTR
    LDY     BRICK_DIG_ROWS,X
    STY     GAME_ROWNUM
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDY     BRICK_DIG_COLS,X
    STY     GAME_COLNUM
    LDA     (PTR1),Y
    CMP     #SPRITE_EMPTY
    BNE     .check_for_brick_fill_player_kill
    JMP     .draw_brick

.check_for_brick_fill_player_kill:
    CMP     #SPRITE_PLAYER
    BNE     .check_for_brick_fill_guard_kill
    LSR     ALIVE

.check_for_brick_fill_guard_kill:
    CMP     #SPRITE_GUARD
    BEQ     .kill_guard

    CMP     #SPRITE_GOLD
    BNE     .draw_brick_
    DEC     GOLD_COUNT

.draw_brick_:
    JMP     .draw_brick

.kill_guard:
    LDA     #SPRITE_BRICK
    STA     (PTR1),Y
    STA     (PTR2),Y
    JSR     DRAW_SPRITE_PAGE1
    LDA     #SPRITE_BRICK
    JSR     DRAW_SPRITE_PAGE2
    LDX     GUARD_COUNT

.find_killed_guard:
    LDA     GUARD_LOCS_COL,X
    CMP     GAME_COLNUM
    BNE     .next_guard
```

```
    LDA     GUARD_LOCS_ROW,X
    CMP     GAME_ROWNUM
    BNE     .next_guard

    LDA     GUARD_GOLD_TIMERS,X
    BPL     .reset_guard_gold_timer
    DEC     GOLD_COUNT

.reset_guard_gold_timer:
    LDA     #$7F
    STA     GUARD_GOLD_TIMERS,X
    STX     GUARD_NUM
    JSR     LOAD_GUARD_DATA
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDX     GUARD_NUM
    LDY     #$01
    STY     GAME_ROWNUM

.row_loop:
    LDY     GAME_ROWNUM
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     GUARD_RESURRECT_COL

.col_loop:
    LDA     (PTR2),Y
    CMP     #$00
    BEQ     .found_good_resurrect_loc

    INC     GUARD_RESURRECT_COL
    LDY     GUARD_RESURRECT_COL
    CPY     #MAX_GAME_COL+1
    BCC     .col_loop

    INC     GAME_ROWNUM
    LDA     #$00
    STA     GUARD_RESURRECT_COL
    BEQ     .row_loop               ; unconditional

.found_good_resurrect_loc:
    TYA
    STA     GUARD_LOCS_COL,X
    LDA     GAME_ROWNUM
    STA     GUARD_LOCS_ROW,X
    LDA     #$14            ; 20
    STA     GUARD_RESURRECTION_TIMERS,X
    LDA     #$02
    STA     GUARD_Y_ADJS,X
    STA     GUARD_X_ADJS,X
    LDA     #$00
```

```
    STA     GUARD_ANIM_STATES,X
    LDY     #$00
    LDA     #$75
    JSR     ADD_AND_UPDATE_SCORE        ; SCORE += 75
    JMP     .next

.next_guard:
    DEX
    BNE     .find_killed_guard

    ; This should never fall through

.draw_brick:
    LDA     #SPRITE_BRICK
    STA     (PTR1),Y
    JSR     DRAW_SPRITE_PAGE1
    LDA     #SPRITE_BRICK
    JSR     DRAW_SPRITE_PAGE2

.next:
    LDX     TMP_LOOP_CTR
    DEX
    BMI     HANDLE_TIMERS_COMMON_RETURN
    JMP     .loop

HANDLE_TIMERS_COMMON_RETURN:
    RTS
```

Defines:
HANDLE_TIMERS, used in chunk 250.
Uses ADD_AND_UPDATE_SCORE 51, ALIVE 111a, DRAW_SPRITE_PAGE1 35, DRAW_SPRITE_PAGE2 35, ERASE_SPRITE_AT_PIXEL_COORDS 38, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_GUARD_SPRITE_AND_COORDS 187b, GET_SCREEN_COORDS_FOR 31a, GOLD_COUNT 81d, GUARD_ANIM_STATES 181, GUARD_COUNT 81d, GUARD_GOLD_TIMERS 181, GUARD_LOCS_COL 181, GUARD_LOCS_ROW 181, GUARD_NUM 181, GUARD_X_ADJS 181, GUARD_Y_ADJS 181, LOAD_GUARD_DATA 186, PTR1 78b, PTR2 78b, SCORE 50b, and TMP_LOOP_CTR 4.

8.2 Startup code

The startup code is run immediately after relocating memory blocks.

129 ⟨*startup code* 129⟩≡ (278)
⟨*set startup softswitches* 130b⟩
⟨*set stack size* 130c⟩
⟨*maybe set carry but not really* 130d⟩
⟨*ready yourself* 131a⟩

The first address, ROMIN_RDROM_WRRAM2 is a bank-select switch. By reading it twice, we set up the memory area from $D000-$DFFF to read from the ROM, but write to RAM bank 2.

The next four softswiches set up the display for full-screen hi-res graphics, page 1.

130a ⟨*defines* 4⟩+≡ (281) ◁122c 136b▷

```
ROMIN_RDROM_WRRAM2      EQU     $C081
TXTCLR                  EQU     $C050
MIXCLR                  EQU     $C052
TXTPAGE1                EQU     $C054
HIRES                   EQU     $C057
```

Defines:
HIRES, used in chunks 124 and 130b.
MIXCLR, used in chunks 124 and 130b.
ROMIN_RDROM_WRRAM2, used in chunk 130b.
TXTCLR, used in chunks 124 and 130b.
TXTPAGE1, used in chunks 73a, 124, 130b, 145, 246, 259, and 268.

130b ⟨*set startup softswitches* 130b⟩≡ (129)

```
        ORG     $5F7D
MAIN:
        LDA     ROMIN_RDROM_WRRAM2
        LDA     ROMIN_RDROM_WRRAM2
        LDA     TXTCLR
        LDA     MIXCLR
        LDA     TXTPAGE1
        LDA     HIRES
```

Defines:
MAIN, used in chunk 275.
Uses HIRES 130a, MIXCLR 130a, ROMIN_RDROM_WRRAM2 130a, TXTCLR 130a, and TXTPAGE1 130a.

The 6502 stack, at maximum, runs from $0100-$01FF. The stack starts at $0100 plus the stack index (the S register), and grows towards $0100. Here we are setting the S register to 0x07 which makes for a very small stack – 8 bytes.

130c ⟨*set stack size* 130c⟩≡ (129)

```
        LDX     #$07
        TXS
```

This next part seems to set the carry only if certain bits in location $5F94 are set. I can find no writes to this location, so the effect is that the carry is cleared. It's entirely possible that this was altered by the cracker.

130d ⟨*maybe set carry but not really* 130d⟩≡ (129)

```
        CLC
        LDA     #$01
        AND     #$A4
        BEQ     .short_delay_mode
        SEC
        ; fall through to .short_delay_mode
```

This next part sets the delay for this game mode, and also reads the keyboard strobe softswtich. That just clears the keyboard strobe in readiness to see if a key is pressed. Then we get dumped into the main loop.

131a ⟨*ready yourself* 131a⟩≡ (129)

```
        ORG     $5F9A

    .short_delay_mode:
        LDX     #$22            ; Number of times to check for keyboard press (34).
        LDY     #$02            ; Number of times to do X checks (2).
                                ; GAME_ROWNUM was initialized to 1, so we do 34*2*1 checks.
        LDA     KBDSTRB
        LDA     #JOYSTICK_MODE
        JMP     CHECK_FOR_BUTTON_DOWN
```

Uses CHECK_FOR_BUTTON_DOWN 131b, GAME_ROWNUM 34a, and KBDSTRB 68b.

Checking for a joystick button (or equivalently the open apple and solid apple keys) to be pressed involves checking the high bit after reading the corresponding button softswitch. Here we're checking if any of the buttons are pressed.

131b ⟨*check for button down* 131b⟩≡ (278)

```
        ORG     $6199

    .poll_inputs:
        LDA     INPUT_MODE

    CHECK_FOR_BUTTON_DOWN:
        CMP     #KEYBOARD_MODE
        BEQ     .poll_keyboard  ; If keyboard mode, skip check button presses.
        LDA     BUTN1
        BMI     .set_level_0_and_play_game
        LDA     BUTN0
        BMI     .set_level_0_and_play_game

        ; fall through to .poll_keyboard
```

Defines:
CHECK_FOR_BUTTON_DOWN, used in chunk 131a.
Uses BUTN0 66, BUTN1 66, and INPUT_MODE 66.

Here we read the keyboard, which involves checking the high bit of the KBD softswitch. This also loads the ASCII code for the key. We check for a keypress in a loop based on the X and Y registers, and on GAME_ROWNUM! So we check for X x Y x GAME_ROWNUM iterations. This controls alternation between "attract-mode" gameplay and the high score screen.

132a ⟨*no button pressed* 132a⟩≡ (278)

```
    ORG     $61A9

.poll_keyboard:
    LDA     KBD
    BMI     .key_pressed
    DEX
    BNE     .poll_inputs
    DEY
    BNE     .poll_inputs
    DEC     GAME_ROWNUM
    BNE     .poll_inputs

    ; fall through to .no_button_or_key_timeout
```

Uses GAME_ROWNUM 34a and KBD 68b.

If one of the joystick buttons was pressed:

132b ⟨*button pressed at startup* 132b⟩≡ (278)

```
    ORG     $6201

.set_level_0_and_play_game:
    LDX     #$00
    STX     DISK_LEVEL_LOC      ; DISK_LEVEL_LOC = 0
    INX
    STX     LEVELNUM            ; LEVELNUM = 1
    STX     $9D
    LDA     #$02
    STA     GAME_MODE
    JMP     INIT_GAME_DATA
```

Uses GAME_MODE 106a and LEVELNUM 52.

And if one of the keys was pressed:

132c ⟨*key pressed at startup* 132c⟩≡ (278)

```
    ORG     $61F6

.key_pressed:
    STA     KBDSTRB     ; Clear keyboard strobe
    CMP     #$85        ; if ctrl-E:
    BEQ     .start_level_editor
    CMP     #$8D        ; if return key:
    BEQ     .read_and_display_hi_score_screen

    ; fall through to .button_pressed
```

Uses KBDSTRB 68b.

Two keys are special, ctrl-E, which opens the level editor, and return, which displays the high score screen.

133a ⟨*ctrl-e pressed* 133a⟩≡ (278)

```
    ORG     $6211

.start_level_editor:
    JMP     LEVEL_EDITOR
```

Uses LEVEL_EDITOR 259.

133b ⟨*return pressed* 133b⟩≡ (278)

```
    ORG     $61E4

.read_and_display_hi_score_screen:
    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read hi score table

    ; fallthrough to .display_hi_score_screen
```

Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231.

Finally, if no key or button was pressed and we've reached the maximum number of polls through the loop:

133c ⟨*timed out waiting for button or keypress* 133c⟩≡ (278)

```
    ORG     $61B8

.no_button_or_key_timeout:
    LDA     GAME_MODE
    BNE     .check_game_mode    ; If GAME_MODE != 0, .check_game_mode.

    ; When GAME_MODE = 0:
    LDX     #$01
    STX     GAME_MODE        ; Set GAME_MODE = 1
    STX     LEVELNUM
    STX     $AC
    STX     $9D                 ; LEVELNUM = $AC = $9D = 1
    LDX     ENABLE_SOUND
    STX     .restore_enable_sound+1    ; Save previous value of DNABLE_SOUND
    STA     ENABLE_SOUND
    JMP     INIT_GAME_DATA

.restore_enable_sound:
    LDA     #$00            ; Fixed up above
    STA     ENABLE_SOUND
    LDA     KBD
    LDX     $AC
    BEQ     .key_pressed
    JMP     LONG_DELAY
```

Uses ENABLE_SOUND 59b, GAME_MODE 106a, KBD 68b, and LEVELNUM 52.

134a ⟨*check game mode* 134a⟩≡ (278)

```
        ORG     $61DE

    .check_game_mode:
        ; For game mode 1, reset and play the game.
        ; For game mode 0, display the high score screen.
        CMP     #$01
        BNE     .reset_game
        BEQ     .display_hi_score_screen        ; Unconditional jump
```

134b ⟨*reset game if not mode 1* 134b⟩≡ (278)

```
        ORG     $61F3

    .reset_game:
        JMP     RESET_GAME
```

Game mode 2 displays the high score screen.

134c ⟨*display high score screen* 134c⟩≡ (278)

```
        ORG     $61E9

    .display_hi_score_screen:
        JSR     HI_SCORE_SCREEN
        LDA     #$02
        STA     GAME_MODE           ; GAME_MODE = 2
        JMP     LONG_DELAY
```

Uses GAME_MODE 106a and HI_SCORE_SCREEN 119b.

When we change over to the high score screen or attract mode, we set the delay to the next mode very large: 195075 times around the loop.

134d ⟨*long delay attract mode* 134d⟩≡ (278)

```
        ORG     $618E

    LONG_DELAY:
        JSR     WAIT_KEY
        LDX     #$FF
        LDY     #$FF
        LDA     #$03
        STA     GAME_ROWNUM

        ; fall through to .poll_inputs
```

Uses GAME_ROWNUM 34a and WAIT_KEY 69a.

8.3 Moving the player

The player's sprite position is stored in PLAYER_COL and PLAYER_ROW, while the offset from the exact sprive location is stored in PLAYER_X_ADJ and PLAYER_Y_ADJ. These adjustments are offset by 2, so that 2 means zero offset. The player also has a PLAYER_ANIM_STATE which is an index into the SPRITE_ANIM_SEQS table. The GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER gets the sprite corresponding to the player's animation state and the player's adjusted screen coordinate.

135a ⟨*tables* 9⟩+≡ (281) ◁121a 138b▷

```
    ORG     $6968
SPRITE_ANIM_SEQS:
    HEX     0B 0C 0D        ; player running left
    HEX     18 19 1A        ; player monkey swinging left
    HEX     0F              ; player digging left
    HEX     13              ; player falling, facing left
    HEX     09 10 11        ; player running right
    HEX     15 16 17        ; player monkey swinging right
    HEX     25              ; player digging right
    HEX     14              ; player falling, facing right
    HEX     0E 12           ; player climbing on ladder
```

Defines:
SPRITE_ANIM_SEQS, used in chunks 84 and 135b.

135b ⟨*get player sprite and coord data* 135b⟩≡ (278)

```
    ORG     $6B85
GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER:
    SUBROUTINE
    ; Using PLAYER_COL/ROW, PLAYER_X/Y_ADJ, and PLAYER_ANIM_STATE,
    ; return the player sprite in A, and the screen coords in X and Y.

    LDX     PLAYER_COL
    LDY     PLAYER_X_ADJ
    JSR     GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR
    STX     SPRITE_NUM          ; Used only as a temporary to save X
    LDY     PLAYER_ROW
    LDX     PLAYER_Y_ADJ
    JSR     GET_SCREEN_ROW_OFFSET_IN_X_FOR
    LDX     PLAYER_ANIM_STATE
    LDA     SPRITE_ANIM_SEQS,X
    LDX     SPRITE_NUM
    RTS
```

Defines:
GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER, used in chunks 43, 157, 159, 161, 164, 168, 171, and 175.
Uses GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR 33c, GET_SCREEN_ROW_OFFSET_IN_X_FOR 33a, PLAYER_ANIM_STATE 84b, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_X_ADJ 84b, PLAYER_Y_ADJ 84b, SPRITE_ANIM_SEQS 135a, and SPRITE_NUM 25c.

Since PLAYER_ANIM_STATE needs to play a sequence over and over, there is a routine to increment the animation state and wrap if necessary. It works by loading A with the lower bound, and X with the upper bound.

136a ⟨*increment player animation state* 136a⟩≡ (278)

```
    ORG     $6BF4
INC_ANIM_STATE:
    SUBROUTINE

    INC     PLAYER_ANIM_STATE
    CMP     PLAYER_ANIM_STATE
    BCC     .check_upper_bound      ; lower bound < PLAYER_ANIM_STATE?
    ; otherwise PLAYER_ANIM_STATE <= lower bound:

.write_lower_bound:
    STA     PLAYER_ANIM_STATE       ; PLAYER_ANIM_STATE = lower bound
    RTS

.check_upper_bound:
    CPX     PLAYER_ANIM_STATE
    BCC     .write_lower_bound       ; PLAYER_ANIM_STATE > upper bound?
    ; otherwise PLAYER_ANIM_STATE <= upper bound:
    RTS
```

Defines:
INC_ANIM_STATE, used in chunks 157, 161, and 164.
Uses PLAYER_ANIM_STATE 84b.

This routine checks whether the player picks up gold. First we check to see if the player's location is exactly on a sprite coordinate, and return if not. Otherwise, we check the background sprite data to see if there's gold at the player's location, and return if not. So if there is gold, we decrement the gold count, put a blank sprite in the background sprite data, increment the score by 250, erase the gold sprite on the background screen at the player location, and then load up data into the sound area.

There is also a flag DIDNT_PICK_UP_GOLD which tells us whether the player did not pick up gold during this move. This flag is set to 1 just before handling the player move.

136b ⟨*defines* 4⟩+≡ (281) ◁130a 138a▷

```
DIDNT_PICK_UP_GOLD      EQU     $94
```

Defines:
DIDNT_PICK_UP_GOLD, used in chunks 43, 137, and 175.

137 ⟨*check for gold picked up by player* 137⟩≡ (278)

```
      ORG     $6B9D
  CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER:
      SUBROUTINE

      LDA     PLAYER_X_ADJ
      CMP     #$02
      BNE     .end
      LDA     PLAYER_Y_ADJ
      CMP     #$02
      BNE     .end

      LDY     PLAYER_ROW
      ⟨set background row pointer PTR2 for Y 78d⟩
      LDY     PLAYER_COL
      LDA     (PTR2),Y

      CMP     #SPRITE_GOLD
      BNE     .end

      LSR     DIDNT_PICK_UP_GOLD  ; picked up gold
      DEC     GOLD_COUNT          ; GOLD_COUNT--

      LDY     PLAYER_ROW
      STY     GAME_ROWNUM
      LDY     PLAYER_COL
      STY     GAME_COLNUM
      LDA     #SPRITE_EMPTY
      STA     (PTR2),Y
      JSR     DRAW_SPRITE_PAGE2   ; Register and draw blank at player loc in background screen

      LDY     GAME_ROWNUM
      LDX     GAME_COLNUM
      JSR     GET_SCREEN_COORDS_FOR
      LDA     #SPRITE_GOLD
      JSR     ERASE_SPRITE_AT_PIXEL_COORDS    ; Erase gold at player loc

      LDY     #$02
      LDA     #$50
      JSR     ADD_AND_UPDATE_SCORE            ; SCORE += 250
      JSR     LOAD_SOUND_DATA
      HEX     07 45 06 55 05 44 04 54 03 43 02 53 00

  .end:
      RTS
```

Defines:
CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER, used in chunks 154, 157, 161, 164, and 175.
Uses ADD_AND_UPDATE_SCORE 51, DIDNT_PICK_UP_GOLD 136b, DRAW_SPRITE_PAGE2 35, ERASE_SPRITE_AT_PIXEL_COORDS 38, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_SCREEN_COORDS_FOR 31a, GOLD_COUNT 81d, LOAD_SOUND_DATA 58, PLAYER_COL 80c,

PLAYER_ROW 80c, PLAYER_X_ADJ 84b, PLAYER_Y_ADJ 84b, PTR2 78b, and SCORE 50b.

138a ⟨*defines* 4⟩+≡ (281) ◁136b 144b▷

```
KEY_COMMAND     EQU     $9E
KEY_COMMAND_LR  EQU     $9F
```

Defines:
KEY_COMMAND, used in chunks 139, 148, 151, 168, 171, 175, and 250.
KEY_COMMAND_LR, used in chunks 139, 148, 151, 168, 171, 175, and 250.

138b ⟨*tables* 9⟩+≡ (281) ◁135a 147▷

```
    ORG     $6B59
VALID_CTRL_KEYS:
    ; ctrl-
    ; ^ @ [ R A S J K H U X Y M
    ; Esc:         ctrl-[
    ; Down arrow:  ctrl-J
    ; Up arrow:    ctrl-K
    ; Right arrow: ctrl-U
    ; Left arrow:  ctrl-H
    ; Return:      ctrl-M
    HEX     9E 80 9B 92 81 93 8A 8B 88 95 98 99 8D 00

    ORG     $6B67
CTRL_KEY_HANDLERS:
    ; These get pushed onto the stack, then an RTS is issued.
    ; Remember that the 6502's return stack contains the address
    ; to return to *minus 1*, so these values are actually one less
    ; than the function to jump to.
    WORD    CTRL_CARET_HANDLER-1
    WORD    CTRL_AT_HANDLER-1
    WORD    ESC_HANDLER-1
    WORD    CTRL_R_HANDLER-1
    WORD    CTRL_A_HANDLER-1
    WORD    CTRL_S_HANDLER-1
    WORD    DOWN_ARROW_HANDLER-1
    WORD    UP_ARROW_HANDLER-1
    WORD    LEFT_ARROW_HANDLER-1
    WORD    RIGHT_ARROW_HANDLER-1
    WORD    CTRL_X_HANDLER-1
    WORD    CTRL_Y_HANDLER-1
    WORD    RETURN_HANDLER-1
```

Defines:
CTRL_KEY_HANDLERS, used in chunk 139.
VALID_CTRL_KEYS, used in chunk 139.
Uses CTRL_A_HANDLER 141c, CTRL_AT_HANDLER 140b, CTRL_CARET_HANDLER 140a, CTRL_R_HANDLER 141c, CTRL_S_HANDLER 142a, CTRL_X_HANDLER 144a, CTRL_Y_HANDLER 144a, DOWN_ARROW_HANDLER 142b, ESC_HANDLER 141b, LEFT_ARROW_HANDLER 143, RETURN_HANDLER 145, RIGHT_ARROW_HANDLER 143, and UP_ARROW_HANDLER 142b.

139 ⟨*check for input* 139⟩≡ (278)

```
        ORG     $6A12
    CHECK_FOR_INPUT:
        SUBROUTINE

        LDA     GAME_MODE
        CMP     #$01
        BEQ     CHECK_FOR_MODE_1_INPUT

        LDX     KBD
        STX     KBDSTRB
        STX     SPRITE_NUM
        BMI     .key_pressed

        LDA     INPUT_MODE
        CMP     #KEYBOARD_MODE
        BEQ     .end                    ; If keyboard mode, end.

    .check_buttons_:
        JMP     READ_JOYSTICK_FOR_COMMAND

    .key_pressed:
        CPX     #$A0
        BCS     .non_ctrl_key_pressed
        ; ctrl key pressed
        STX     SPRITE_NUM
        LDY     #$FF

    .loop:
        INY
        LDA     VALID_CTRL_KEYS,Y
        BEQ     .non_ctrl_key_pressed

        CMP     SPRITE_NUM
        BNE     .loop

        TYA
        ASL
        TAY
        LDA     CTRL_KEY_HANDLERS+1,Y
        PHA
        LDA     CTRL_KEY_HANDLERS,Y
        PHA
        RTS                             ; JSR to CTRL_KEY_HANDLERS[Y], then return.

    .non_ctrl_key_pressed:
        LDA     INPUT_MODE
        CMP     #JOYSTICK_MODE
        BEQ     .check_buttons_         ; If joystick mode, check buttons.
```

```
    LDX     SPRITE_NUM
    STX     KEY_COMMAND
    STX     KEY_COMMAND_LR

.end:
    RTS
```

Defines:
  CHECK_FOR_INPUT, used in chunks 140–45 and 175.
Uses CHECK_FOR_MODE_1_INPUT 148, CTRL_KEY_HANDLERS 138b, GAME_MODE 106a, INPUT_MODE 66, KBD 68b, KBDSTRB 68b, KEY_COMMAND 138a, KEY_COMMAND_LR 138a, READ_JOYSTICK_FOR_COMMAND 151, SPRITE_NUM 25c, and VALID_CTRL_KEYS 138b.

Hitting `ctrl-^` increments both lives and level number, but also kills the player.

140a ⟨*ctrl handlers* 140a⟩≡ (278) 140b▷

```
    ORG     $6A56
CTRL_CARET_HANDLER:
    SUBROUTINE

    INC     LIVES
    INC     LEVELNUM
    INC     DISK_LEVEL_LOC
    LSR     ALIVE       ; set player dead
    LSR     $9D
    RTS
```

Defines:
  CTRL_CARET_HANDLER, used in chunk 138b.
Uses ALIVE 111a, LEVELNUM 52, and LIVES 52.

Hitting `ctrl-@` increments lives.

140b ⟨*ctrl handlers* 140a⟩+≡ (278) ◁140a 141b▷

```
    ORG     $6A61
CTRL_AT_HANDLER:
    SUBROUTINE

    INC     LIVES
    BNE     .have_lives
    DEC     LIVES           ; LIVES = 255
.have_lives:
    JSR     PUT_STATUS_LIVES
    LSR     $9D
    JMP     CHECK_FOR_INPUT
```

Defines:
  CTRL_AT_HANDLER, used in chunk 138b.
Uses CHECK_FOR_INPUT 139, LIVES 52, and PUT_STATUS_LIVES 53.

There's some dead code which seems to increase the GUARD_PATTERN_OFFSET and then kill the player, but not remove a life.

141a ⟨*dead code* 116⟩+≡ (281) ◁116 277a▷

```
    ORG     $6A6F
INC_GUARD_PATTERN_OFFSET:
    SUBROUTINE

    INC     GUARD_PATTERN_OFFSET
    INC     LIVES
    LSR     ALIVE
    RTS
```

Defines:
INC_GUARD_PATTERN_OFFSET, never used.
Uses ALIVE 111a, GUARD_PATTERN_OFFSET 245c, and LIVES 52.

Hitting ESC pauses the game, and ESC then unpauses the game.

141b ⟨*ctrl handlers* 140a⟩+≡ (278) ◁140b 141c▷

```
    ORG     $6A76
ESC_HANDLER:
    SUBROUTINE

    JSR     WAIT_KEY_QUEUED
    CMP     #$9B            ; key pressed is ESC?
    BNE     ESC_HANDLER
    JMP     CHECK_FOR_INPUT
```

Defines:
ESC_HANDLER, used in chunk 138b.
Uses CHECK_FOR_INPUT 139 and WAIT_KEY_QUEUED 69b.

Hitting ctrl-R sets lives to 1 and sets player to dead, ending the game. Hitting ctrl-A shifts ALIVE, which just kills you.

141c ⟨*ctrl handlers* 140a⟩+≡ (278) ◁141b 142a▷

```
    ORG     $6A80
CTRL_R_HANDLER:
    SUBROUTINE

    LDA     #$01
    STA     LIVES

CTRL_A_HANDLER:
    LSR     ALIVE           ; Set player to dead
    RTS
```

Defines:
CTRL_A_HANDLER, used in chunk 138b.
CTRL_R_HANDLER, used in chunk 138b.
Uses ALIVE 111a and LIVES 52.

Hitting `ctrl-S` toggles sound.

142a ⟨*ctrl handlers* 140a⟩+≡ (278) ◁141c 142b▷

```
    ORG     $6A87
CTRL_S_HANDLER:
    SUBROUTINE

    LDA     ENABLE_SOUND
    EOR     #$FF
    STA     ENABLE_SOUND
    JMP     CHECK_FOR_INPUT
```

Defines:
  CTRL_S_HANDLER, used in chunk 138b.
Uses CHECK_FOR_INPUT 139 and ENABLE_SOUND 59b.

Hitting `ctrl-J` switches to joystick controls, and hitting `ctrl-K` switches to keyboard controls.

142b ⟨*ctrl handlers* 140a⟩+≡ (278) ◁142a 143▷

```
    ORG     $6A90
DOWN_ARROW_HANDLER:
    SUBROUTINE

    LDA     #JOYSTICK_MODE
    STA     INPUT_MODE
    JMP     CHECK_FOR_INPUT

    ORG     $6A97
UP_ARROW_HANDLER:
    SUBROUTINE

    LDA     #KEYBOARD_MODE
    STA     INPUT_MODE
    JMP     CHECK_FOR_INPUT
```

Defines:
  DOWN_ARROW_HANDLER, used in chunk 138b.
  UP_ARROW_HANDLER, used in chunk 138b.
Uses CHECK_FOR_INPUT 139 and INPUT_MODE 66.

Hitting the left arrow and right arrow decreases and increases the FRAME_PERIOD, effectively speed up and slowing down the game.

143 ⟨*ctrl handlers* 140a⟩+≡ (278) ◁142b 144a▷

```
    ORG     $6ABC
RIGHT_ARROW_HANDLER:
    SUBROUTINE

    LDA     FRAME_PERIOD
    BEQ     LEFT_ARROW_HANDLER_end
    DEC     FRAME_PERIOD
    JMP     CHECK_FOR_INPUT

    ORG     $6AC5
LEFT_ARROW_HANDLER:
    SUBROUTINE

    LDA     FRAME_PERIOD
    CMP     #$0F
    BEQ     LEFT_ARROW_HANDLER_end
    INC     FRAME_PERIOD

LEFT_ARROW_HANDLER_end:
    JMP     CHECK_FOR_INPUT
```

Defines:
LEFT_ARROW_HANDLER, used in chunk 138b.
RIGHT_ARROW_HANDLER, used in chunk 138b.
Uses CHECK_FOR_INPUT 139 and FRAME_PERIOD 61b.

Hitting ctrl-X reverses one axis of the joystick, while hitting ctrl-Y reverses the other axis. These thresholds are used by READ_JOYSTICK_FOR_COMMAND.

144a ⟨*ctrl handlers* 140a⟩+≡ (278) ◁143

```
    ORG     $6A9E
CTRL_X_HANDLER:
    SUBROUTINE

    LDA     PADDLE0_THRESH1
    LDX     PADDLE0_THRESH2
    STA     PADDLE0_THRESH2
    STX     PADDLE0_THRESH1
    JMP     CHECK_FOR_INPUT

    ORG     $6AAD
CTRL_Y_HANDLER:
    SUBROUTINE

    LDA     PADDLE1_THRESH1
    LDX     PADDLE1_THRESH2
    STA     PADDLE1_THRESH2
    STX     PADDLE1_THRESH1
    JMP     CHECK_FOR_INPUT
```

Defines:
CTRL_X_HANDLER, used in chunk 138b.
CTRL_Y_HANDLER, used in chunk 138b.
Uses CHECK_FOR_INPUT 139, PADDLE0_THRESH1 150, PADDLE0_THRESH2 150, PADDLE1_THRESH1 150, and PADDLE1_THRESH2 150.

144b ⟨*defines* 4⟩+≡ (281) ◁138a 166▷

```
BRICK_DIG_COLS      EQU     $0CA0       ; 31 bytes of col nums
BRICK_DIG_ROWS      EQU     $0CC0       ; 31 bytes of row nums
BRICK_FILL_TIMERS   EQU     $0CE0       ; 31 bytes of fill timers
```

145 ⟨*return handler* 145⟩≡ (278)

```
        ORG     $77AC
    RETURN_HANDLER:
        SUBROUTINE

        JSR     HI_SCORE_SCREEN     ; show high score screen
        LDX     #$FF
        LDY     #$FF
        LDA     #$04
        STA     SCRATCH_A1          ; loop 256x256x4 times

    .loop:
        LDA     INPUT_MODE
        CMP     #KEYBOARD_MODE              ; Keyboard mode
        BEQ     .check_keyboard

        LDA     BUTN1
        BMI     .button_pressed
        LDA     BUTN0
        BMI     .button_pressed

    .check_keyboard:
        LDA     KBD
        BMI     .button_pressed

        DEX
        BNE     .loop
        DEY
        BNE     .loop
        DEC     SCRATCH_A1
        BNE     .loop

    .button_pressed:
        STA     KBDSTRB
        STA     TXTPAGE1
        JSR     CLEAR_HGR2
        LDY     #MAX_GAME_ROW
        STY     GAME_ROWNUM

    .loop2:
        ⟨set background row pointer PTR2 for Y 78d⟩
        LDY     #MAX_GAME_COL
        STY     GAME_COLNUM

    .loop3:
        LDA     (PTR2),Y
        CMP     #SPRITE_TRAP
        BNE     .draw_sprite
        LDA     #SPRITE_BRICK
    .draw_sprite:
```

```
    JSR     DRAW_SPRITE_PAGE2

    DEC     GAME_COLNUM
    LDY     GAME_COLNUM
    BPL     .loop3

    DEC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    BPL     .loop2

    LDX     #$1E
.loop4:
    STX     TMP_LOOP_CTR
    LDA     BRICK_FILL_TIMERS,X
    BEQ     .next4

    LDY     BRICK_DIG_ROWS,X
    STY     GAME_ROWNUM
    LDY     BRICK_DIG_COLS,X
    STY     GAME_COLNUM
    CMP     #$15
    BCC     .check_b

    LDA     #SPRITE_EMPTY
    JSR     DRAW_SPRITE_PAGE2
    JMP     .next4

.check_b:
    CMP     #$0B
    BCC     .draw_sprite_56
    LDA     #$37
    JSR     DRAW_SPRITE_PAGE2
    JMP     .next4

.draw_sprite_56:
    LDA     #$38
    JSR     DRAW_SPRITE_PAGE2

.next4:
    LDX     TMP_LOOP_CTR
    DEX
    BPL     .loop4

    LDX     GUARD_COUNT
    BEQ     .check_for_input

.loop5:
    LDA     GUARD_RESURRECTION_TIMERS,X
    STX     TMP_LOOP_CTR
    BEQ     .next5
```

```
    LDY     GUARD_LOCS_COL,X
    STY     GAME_COLNUM
    LDY     GUARD_LOCS_ROW,X
    STY     GAME_ROWNUM
    CMP     #$14
    BCS     .next5

    CMP     #$0B
    BCC     .draw_sprite_58
    LDA     #$39                ; sprite 57
    BNE     .draw_sprite2       ; unconditional

.draw_sprite_58:
    LDA     #$3A

.draw_sprite2:
    JSR     DRAW_SPRITE_PAGE2

.next5:
    LDX     TMP_LOOP_CTR
    DEX
    BNE     .loop5

.check_for_input:
    JMP     CHECK_FOR_INPUT
```

Defines:
RETURN_HANDLER, used in chunk 138b.
Uses BUTN0 66, BUTN1 66, CHECK_FOR_INPUT 139, CLEAR_HGR2 5, DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, GAME_ROWNUM 34a, GUARD_COUNT 81d, GUARD_LOCS_COL 181, GUARD_LOCS_ROW 181, HI_SCORE_SCREEN 119b, INPUT_MODE 66, KBD 68b, KBDSTRB 68b, PTR2 78b, SCRATCH_A1 4, TMP_LOOP_CTR 4, and TXTPAGE1 130a.

During pregame mode 1, we don't check for gameplay input. Instead, we use CHECK_FOR_MODE_1_INPUT for input. We first check if the user has pressed a key or hit a joystick button, and if so, we simulate killing the attract-mode player. However, if nothing was pressed, we check if the simulated player is pressing a key, and handle that.

147 ⟨*tables* 9⟩+≡ (281) ◁138b 150▷

```
    ORG     $6A0B
VALID_KEY_COMMANDS:
    HEX     C9      ; 'I'
    HEX     CA      ; 'J'
    HEX     CB      ; 'K'
    HEX     CC      ; 'L'
    HEX     CF      ; 'O'
    HEX     D5      ; 'U'
    HEX     A0      ; space
```

Defines:
VALID_KEY_COMMANDS, used in chunk 148.

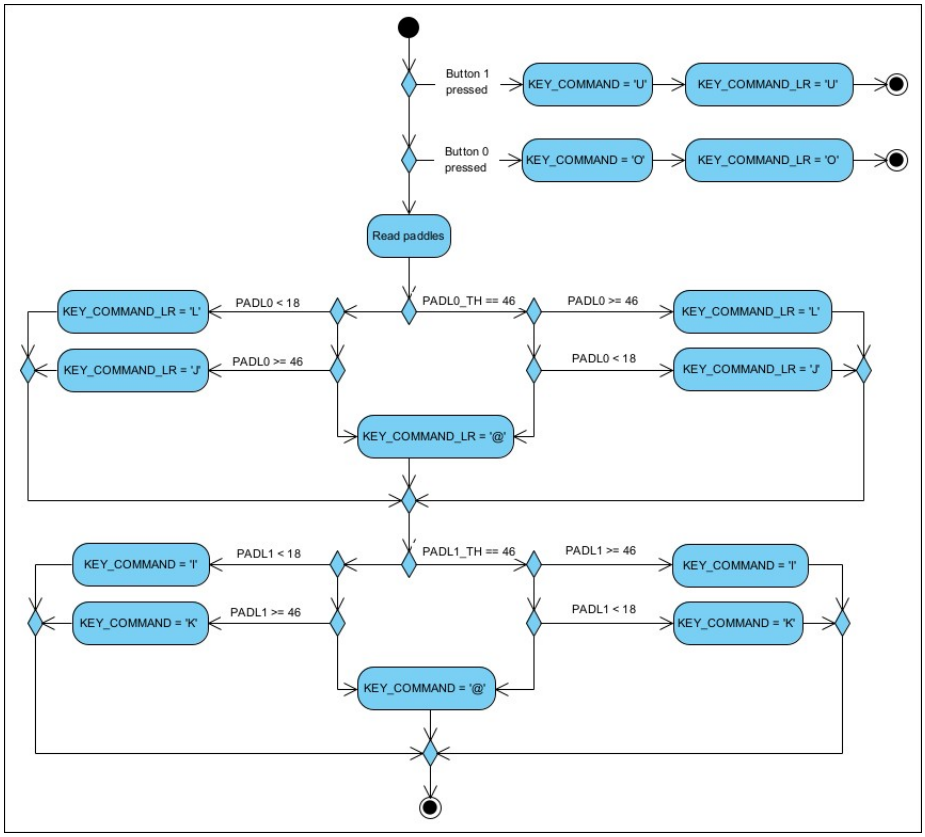
148 ⟨*check for mode 1 input* 148⟩≡ (278)

```
        ORG     $69B8
    CHECK_FOR_MODE_1_INPUT:
        SUBROUTINE

        LDA     KBD
        BMI     .key_pressed

        LDA     INPUT_MODE
        CMP     #KEYBOARD_MODE
        BEQ     .nothing_pressed

        ; Check joystick buttons also
        LDA     BUTN1
        BMI     .key_pressed
        LDA     BUTN0
        BPL     .nothing_pressed

    .key_pressed:
        ; Simulate killing the attact-mode player.
        LSR     $AC
        LSR     ALIVE
        LDA     #$01
        STA     LIVES
        RTS

    .nothing_pressed:
        LDA     $AB
        BNE     .sim_keypress

        LDY     #$00
        LDA     ($A8),Y
        STA     $AA
        INY
        LDA     ($A8),Y
        STA     $AB

        ; {$A8,$A9} += 2
        LDA     $A8
        CLC
        ADC     #$02
        STA     $A8
        LDA     $A9
        ADC     #$00
        STA     $A9

    .sim_keypress:
        LDA     $AA
        AND     #$0F
        TAX
```

```
    LDA     VALID_KEY_COMMANDS,X
    STA     KEY_COMMAND
    LDA     $AA
    LSR
    LSR
    LSR
    LSR
    TAX
    LDA     VALID_KEY_COMMANDS,X
    STA     KEY_COMMAND_LR
    DEC     $AB
    RTS
```

Defines:
CHECK_FOR_MODE_1_INPUT, used in chunk 139.
Uses ALIVE 111a, BUTN0 66, BUTN1 66, INPUT_MODE 66, KBD 68b, KEY_COMMAND 138a, KEY_COMMAND_LR 138a, LIVES 52, and VALID_KEY_COMMANDS 147.

The READ_JOYSTICK_FOR_COMMAND routine reads the paddle buttons and joystick state. If a paddle button is pressed, it translates to the dig left or dig right keyboard command. Otherwise, the joystick state is read and compared to the thresholds 0x12 and 0x2E, which can be reversed through the ctrl-X key, and translated to left, right, up, or down keyboard commands.



150 ⟨*tables* 9⟩+≡ (281) ◁147 167▷

```
    ORG     $6B81
PADDLE0_THRESH1:
    HEX     12
PADDLE0_THRESH2:
    HEX     2E
PADDLE1_THRESH1:
    HEX     12
PADDLE1_THRESH2:
    HEX     2E
```

Defines:
PADDLE0_THRESH1, used in chunks 144a and 151.
PADDLE0_THRESH2, used in chunks 144a and 151.
PADDLE1_THRESH1, used in chunks 144a and 151.
PADDLE1_THRESH2, used in chunks 144a and 151.

151 ⟨*check buttons* 151⟩≡ (278)

```
    ORG     $6AD0
READ_JOYSTICK_FOR_COMMAND:
    SUBROUTINE

    LDA     BUTN1
    BPL     .check_butn0
    LDA     #$D5                ; 'U' (dig)
    BNE     .store_key_command  ; unconditional

.check_butn0:
    LDA     BUTN0
    BPL     .read_paddles
    LDA     #$CF                ; 'O' (dig)

.store_key_command
    STA     KEY_COMMAND
    STA     KEY_COMMAND_LR
    RTS

.read_paddles:
    JSR     READ_PADDLES
    LDY     PADDLE0_VALUE

    LDA     PADDLE0_THRESH2
    CMP     #$2E
    BEQ     .6afa
    ; PADDLE0_THRESH2 != 46

    CPY     PADDLE0_THRESH2
    BCS     .6b03
    ; PADDLE0_VALUE < PADDLE0_THRESH2

    LDA     #$CC                ; 'L' (right)
    BNE     .check_paddle_1     ; unconditional

.6afa:
    CPY     PADDLE0_THRESH2
    BCC     .6b03
    ; PADDLE0_VALUE >= PADDLE0_THRESH2

    LDA     #$CC                ; 'L' (right)
    BNE     .check_paddle_1     ; unconditional

.6b03:
    LDA     PADDLE0_THRESH1
    CMP     #$2E
    BEQ     .6b13
    ; PADDLE0_THRESH1 != 46
```

```
    CPY     PADDLE0_THRESH1
    BCS     .6b1c
    ; PADDLE0_VALUE < PADDLE0_THRESH1
    LDA     #$CA                ; 'J' (left)
    BNE     .check_paddle_1     ; unconditional

.6b13:
    CPY     PADDLE0_THRESH1
    BCC     .6b1c
    ; PADDLE0_VALUE >= PADDLE0_THRESH1
    LDA     #$CA                ; 'J' (left)
    BNE     .check_paddle_1     ; unconditional

.6b1c:
    LDA     #$C0                ; '@'

.check_paddle_1:
    STA     KEY_COMMAND_LR

    LDY     PADDLE1_VALUE

    LDA     PADDLE1_THRESH1
    CMP     #$2E
    BEQ     .6b32
    ; PADDLE1_THRESH1 != 46

    CPY     PADDLE1_THRESH1
    BCS     .6b3b
    ; PADDLE1_VALUE >= PADDLE1_THRESH1

    LDA     #$C9                        ; 'I' (up)
    BNE     .store_key_command_end      ; unconditional

.6b32:
    CPY     PADDLE1_THRESH1
    BCC     .6b3b
    ; PADDLE1_VALUE < PADDLE1_THRESH1

    LDA     #$C9                        ; 'I' (up)
    BNE     .store_key_command_end      ; unconditional

.6b3b:
    LDA     PADDLE1_THRESH2
    CMP     #$2E
    BEQ     .6b4b
    ; PADDLE1_THRESH2 != 46

    CPY     PADDLE1_THRESH2
    BCS     .6b54
    ; PADDLE1_VALUE >= PADDLE1_THRESH2
```

```
    LDA     #$CB                        ; 'K' (down)
    BNE     .store_key_command_end      ; unconditional

.6b4b:
    CPY     PADDLE1_THRESH2
    BCC     .6b54
    ; PADDLE1_VALUE < PADDLE1_THRESH2

    LDA     #$CB                        ; 'K' (down)
    BNE     .store_key_command_end      ; unconditional

.6b54:
    LDA     #$C0                        ; '@'

.store_key_command_end:
    STA     KEY_COMMAND
    RTS
```

Defines:
READ_JOYSTICK_FOR_COMMAND, used in chunk 139.
Uses BUTN0 66, BUTN1 66, KEY_COMMAND 138a, KEY_COMMAND_LR 138a, PADDLE0_THRESH1 150, PADDLE0_THRESH2 150, PADDLE0_VALUE 64, PADDLE1_THRESH1 150, PADDLE1_THRESH2 150, PADDLE1_VALUE 64, and READ_PADDLES 65.

8.4 Player movement

Player movement is generally handled by functions which check whether the player can move in a given direction, and then either fail with carry set, or succeed, and the player is moved, with carry cleared.

Recall that the player is at the gross sprite location given by PLAYER_COL and PLAYER_ROW, but with a plus-or-minus adjustment given by a horizontal adjustment PLAYER_X_ADJ and a vertical adjustment PLAYER_Y_ADJ.

We will refer to the player as "exactly on" the sprite if the adjustment in the direction we're interested in is zero. Again, recall that the adjustment values are offset by 2, so an adjustment of zero is a value of 2, and the adjustment ranges from -2 to +2.

We can refer to the player as slightly above, below, left of, or right of, an exact sprite coordinate if the adjustment is not zero.

There are two routines which nudge the player towards an exact sprite row or column. Generally this is done when the player does something that has to take place on an exact row or column, such as climbing a ladder or traversing a rope, and serves to make the transition to an aligned row or column more smooth. Each time the player is nudged, we also check if the player landed on gold.

154 ⟨*try moving up* 154⟩≡ (278) 157▷

```
    ORG     $6C13
NUDGE_PLAYER_TOWARDS_EXACT_COLUMN:
    SUBROUTINE

    LDA     PLAYER_X_ADJ
    CMP     #$02
    BCC     .player_slightly_left
    BEQ     .end

.player_slightly_right:
    DEC     PLAYER_X_ADJ        ; Nudge player left
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.player_slightly_left:
    INC     PLAYER_X_ADJ        ; Nudge player right
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.end:
    RTS

    ORG     $6C26
NUDGE_PLAYER_TOWARDS_EXACT_ROW:
    SUBROUTINE

    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BCC     .player_slightly_above
```

```
    BEQ     .end

.player_slightly_below:
    DEC     PLAYER_Y_ADJ        ; Nudge player up
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.player_slightly_above:
    INC     PLAYER_Y_ADJ        ; Nudge player down
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.end:
    RTS
```

Defines:
NUDGE_PLAYER_TOWARDS_EXACT_COLUMN, used in chunks 157, 159, 168, 171, and 175.
NUDGE_PLAYER_TOWARDS_EXACT_ROW, used in chunks 161, 164, 168, and 171.
Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 137, PLAYER_X_ADJ 84b, and PLAYER_Y_ADJ 84b.

Now the logic for attempting to move up is:

- If the player location contains a ladder:
  - If the player is slightly below the sprite, then move the player up.
  - Otherwise, if the player is on row zero, the player cannot move up.
  - Otherwise, if the sprite on the row above is brick, stone, or trap, the player cannot move up.
  - Otherwise, the player can move up.
- Otherwise:
  - If the player is not slightly below the sprite, the player cannot move up.
  - Otherwise, if the sprite on the row below is not a ladder, the player cannot move up.
  - Otherwise, the player can move up.

The steps involved in actually moving the player up are:

- Erase the player sprite.
- Reduce any horizontal adjustment, checking for gold pickup if not already exactly on a sprite column.
- Adjust the player vertically upwards by decrementing `PLAYER_Y_ADJ`.
- If the adjustment didn't roll over, check for gold pickup, then update the player animation for climbing, and draw the player.
- Otherwise:
  - Copy the background sprite at the player's sprite location to the active page, unless that sprite is a brick, in which case place an empty on the active page.
  - Decrement `PLAYER_ROW`.
  - Put the player sprite on the active page at the new location.
  - Set the player's vertical adjustment to `+2`.
  - Update the player animation for climbing, and draw the player.

157 ⟨*try moving up* 154⟩+≡ (278) ◁154

```
        ORG     $66BD
    TRY_MOVING_UP:
        SUBROUTINE

        ⟨get background sprite at player location 80e⟩
        CMP     #SPRITE_LADDER
        BEQ     .ladder_here

        LDY     PLAYER_Y_ADJ
        CPY     #$03
        BCC     .cannot_move       ; if PLAYER_Y_ADJ <= 2

        ; and if there's no ladder below, you can't move up.
        ⟨get background sprite at player location on next row 80f⟩
        CMP     #SPRITE_LADDER
        BEQ     .move_player_up

    .cannot_move:
        SEC
        RTS

    .ladder_here:
        LDY     PLAYER_Y_ADJ
        CPY     #$03
        BCS     .move_player_up             ; if PLAYER_Y_ADJ > 2

        ; If you're at the top, you can't move up even if there's a ladder.
        LDY     PLAYER_ROW
        BEQ     .cannot_move       ; if PLAYER_ROW == 0, set carry and return

        ; You can't move up if there's a brick, stone, or trap above.
        LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS-1,Y
        STA     PTR1
        LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES-1,Y
        STA     PTR1+1
        LDY     PLAYER_COL
        LDA     (PTR1),Y                    ; Get the sprite on the row above.

        CMP     #SPRITE_BRICK
        BEQ     .cannot_move
        CMP     #SPRITE_STONE
        BEQ     .cannot_move
        CMP     #SPRITE_TRAP
        BEQ     .cannot_move       ; If brick, stone, or trap, set carry and return

    .move_player_up:
        JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
        JSR     ERASE_SPRITE_AT_PIXEL_COORDS
        LDY     PLAYER_ROW
```

```
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_COLUMN
    DEC     PLAYER_Y_ADJ                ; Move player up
    BPL     TRY_MOVING_UP_check_for_gold

    ; PLAYER_Y_ADJ rolled over.

    ; Restore the sprite at the player's former location:
    ; If background page at player location is brick, put an empty at the
    ; (previous) player location on active page, otherwise copy the background
    ; sprite to the active page.
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BNE     .set_on_real_page
    LDA     #SPRITE_EMPTY
.set_on_real_page:
    STA     (PTR1),Y

    DEC     PLAYER_ROW                  ; Move player up
    LDY     PLAYER_ROW
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     PLAYER_COL
    LDA     #SPRITE_PLAYER
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #$04
    STA     PLAYER_Y_ADJ        ; Set adjustment to +2
    BNE     TRY_MOVING_UP_inc_anim_state     ; unconditional

TRY_MOVING_UP_check_for_gold:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

TRY_MOVING_UP_inc_anim_state:
    LDA     #$10
    LDX     #$11
    JSR     INC_ANIM_STATE      ; player climbing on ladder
    JSR     DRAW_PLAYER
    CLC
    RTS
```

Defines:
TRY_MOVING_UP, used in chunk 175.
Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 137, CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 78a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 78a, DRAW_PLAYER 43, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, INC_ANIM_STATE 136a, NUDGE_PLAYER_TOWARDS_EXACT_COLUMN 154, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_Y_ADJ 84b, PTR1 78b, and PTR2 78b.

For attempting to move down, the logic is:

- If the player is slightly above the sprite, then move the player down.
- Otherwise, if the player is on row 15 or more, the player cannot move down.
- Otherwise, if the row below is stone or brick, the player cannot move down.
- Otherwise, the player can move down.

The steps involved in actually moving the player down are:

- Erase the player sprite.
- Reduce any horizontal adjustment, checking for gold pickup if not already exactly on a sprite column.
- Adjust the player vertically downwards by incrementing `PLAYER_Y_ADJ`.
- If the adjustment didn't roll over, check for gold pickup, then update the player animation for climbing, and draw the player.
- Otherwise:
  - Copy the background sprite at the player's sprite location to the active page, unless that sprite is a brick, in which case place an empty on the active page.
  - Increment `PLAYER_ROW`.
  - Put the player sprite on the active page at the new location.
  - Set the player's vertical adjustment to `-2`.
  - Update the player animation for climbing, and draw the player.

159 ⟨*try moving down* 159⟩≡ (278)

```
    ORG     $6766
TRY_MOVING_DOWN:
    SUBROUTINE

    LDY     PLAYER_Y_ADJ
    CPY     #$02
    BCC     .move_player_down   ; player slightly above, so can move down.

    LDY     PLAYER_ROW
    CPY     #MAX_GAME_ROW
    BCS     .cannot_move        ; player on row >= 15, so cannot move.

    ⟨set active row pointer PTR1 for Y+1 79d⟩
    LDY     PLAYER_COL
    LDA     (PTR1),Y
    CMP     #SPRITE_STONE
```

```
    BEQ     .cannot_move
    CMP     #SPRITE_BRICK
    BNE     .move_player_down   ; Row below is stone or brick, so cannot move.

.cannot_move:
    SEC
    RTS

.move_player_down:
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_COLUMN
    INC     PLAYER_Y_ADJ                ; Move player down
    LDA     PLAYER_Y_ADJ
    CMP     #$05
    BCC     .check_for_gold_

    ; adjustment overflow
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BNE     .set_on_real_page
    LDA     #SPRITE_EMPTY
.set_on_real_page:
    STA     (PTR1),Y

    INC     PLAYER_ROW
    LDY     PLAYER_ROW
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     PLAYER_COL
    LDA     #SPRITE_PLAYER
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #SPRITE_EMPTY
    STA     PLAYER_Y_ADJ        ; Set adjustment to -2
    JMP     TRY_MOVING_UP_inc_anim_state

.check_for_gold_:
    JMP     TRY_MOVING_UP_check_for_gold
```

Defines:
TRY_MOVING_DOWN, used in chunk 175.
Uses ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, NUDGE_PLAYER_TOWARDS_EXACT_COLUMN 154, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_Y_ADJ 84b, PTR1 78b, and PTR2 78b.

For attempting to move left, the logic is:

- If the player is slightly right of the sprite, then move the player left.
- Otherwise, if the player is on column 0, the player cannot move left.
- Otherwise, if the column to the left is stone, brick, or trap, the player cannot move left.
- Otherwise, the player can move left.

The steps involved in actually moving the player left are:

- Erase the player sprite.
- Set the PLAYER_FACING_DIRECTION to left (0xFF).
- Reduce any vertical adjustment, checking for gold pickup if not already exactly on a sprite column.
- Adjust the player horizontally to the left by decrementing PLAYER_X_ADJ.
- If the adjustment didn't roll over, check for gold pickup, then update the player animation for moving left, and draw the player.
- Otherwise:
  - Copy the background sprite at the player's sprite location to the active page, unless that sprite is a brick, in which case place an empty on the active page.
  - Decrement PLAYER_COL.
  - Put the player sprite on the active page at the new location.
  - Set the player's horizontal adjustment to +2.
  - Update the player animation for moving left, and draw the player.

The animation is either monkey-traversing if the player moves onto a rope, or running otherwise.

161 ⟨*try moving left* 161⟩≡ (278)

```
    ORG     $65D3
TRY_MOVING_LEFT:
    SUBROUTINE

    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDX     PLAYER_X_ADJ
    CPX     #$03
    BCS     .move_player_left       ; player slightly right, so can move left.

    LDY     PLAYER_COL
```

```
    BEQ     .cannot_move            ; col == 0, so cannot move.

    DEY
    LDA     (PTR1),Y
    CMP     #SPRITE_STONE
    BEQ     .cannot_move
    CMP     #SPRITE_BRICK
    BEQ     .cannot_move
    CMP     #SPRITE_TRAP
    BNE     .move_player_left       ; brick, stone, or trap to left, so cannot move.

.cannot_move:
    RTS

.move_player_left:
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDA     #$FF
    STA     PLAYER_FACING_DIRECTION             ; face left
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_ROW
    DEC     PLAYER_X_ADJ
    BPL     .check_for_gold

    ; adjustment overflow
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BNE     .set_on_level
    LDA     #SPRITE_EMPTY
.set_on_level:
    STA     (PTR1),Y

    DEC     PLAYER_COL
    DEY
    LDA     #SPRITE_PLAYER
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #$04
    STA     PLAYER_X_ADJ        ; Set adjustment to +2
    BNE     .inc_anim_state     ; Unconditional

.check_for_gold:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.inc_anim_state:
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_ROPE
    BEQ     .anim_state_monkeying

    LDA     #$00
```

```
    LDX     #$02
    BNE     .done               ; Unconditional

.anim_state_monkeying:
    LDA     #$03
    LDX     #$05

.done:
    JSR     INC_ANIM_STATE
    JMP     DRAW_PLAYER
```

Defines:
TRY_MOVING_LEFT, used in chunk 175.
Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 137, DRAW_PLAYER 43, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, INC_ANIM_STATE 136a, NUDGE_PLAYER_TOWARDS_EXACT_ROW 154, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_X_ADJ 84b, PTR1 78b, and PTR2 78b.

Moving right has the same logic as moving left, except in the other direction.

164 ⟨*try moving right* 164⟩≡ (278)

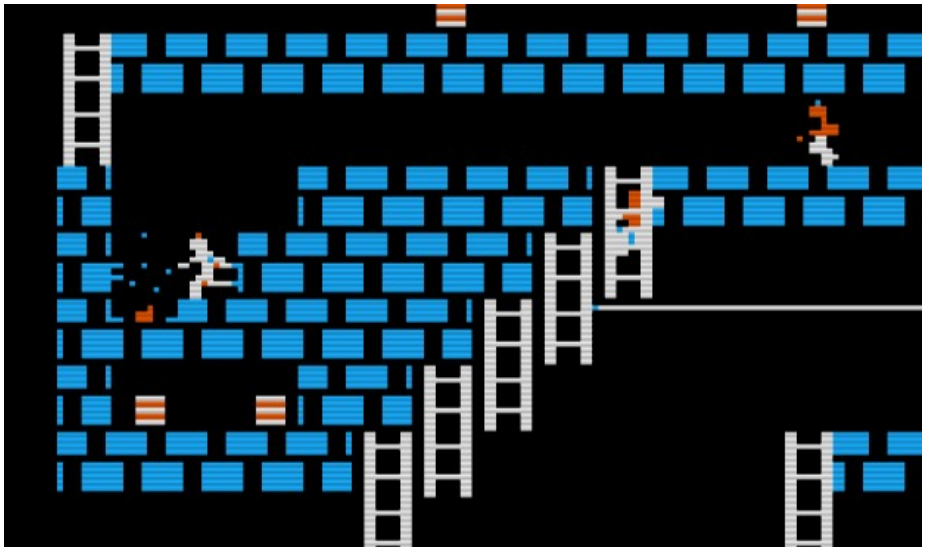
```
    ORG     $6645
TRY_MOVING_RIGHT:
    SUBROUTINE

    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDX     PLAYER_X_ADJ
    CPX     #$02
    BCC     .move_player_right      ; player slightly left, so can move right.

    LDY     PLAYER_COL
    CPY     #MAX_GAME_COL
    BEQ     .cannot_move            ; col == 27, so cannot move.

    INY
    LDA     (PTR1),Y
    CMP     #SPRITE_STONE
    BEQ     .cannot_move
    CMP     #SPRITE_BRICK
    BEQ     .cannot_move
    CMP     #SPRITE_TRAP
    BNE     .move_player_right      ; brick, stone, or trap to right, so cannot move.

.cannot_move:
    RTS

.move_player_right:
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDA     #$01
    STA     PLAYER_FACING_DIRECTION             ; face right
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_ROW
    INC     PLAYER_X_ADJ
    LDA     PLAYER_X_ADJ
    CMP     #$05
    BCC     .check_for_gold

    ; adjustment overflow
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BNE     .set_on_level
    LDA     #SPRITE_EMPTY
.set_on_level:
    STA     (PTR1),Y

    INC     PLAYER_COL
```

```
    INY
    LDA     #SPRITE_PLAYER
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #$00
    STA     PLAYER_X_ADJ        ; Set adjustment to -2
    BEQ     .inc_anim_state     ; Unconditional

.check_for_gold:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.inc_anim_state:
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_ROPE
    BEQ     .anim_state_monkeying

    LDA     #$08
    LDX     #$0A
    BNE     .done               ; Unconditional

.anim_state_monkeying:
    LDA     #$0B
    LDX     #$0D

.done:
    JSR     INC_ANIM_STATE
    JMP     DRAW_PLAYER
```

Defines:
TRY_MOVING_RIGHT, used in chunk 175.
Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 137, DRAW_PLAYER 43, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, INC_ANIM_STATE 136a, NUDGE_PLAYER_TOWARDS_EXACT_ROW 154, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_X_ADJ 84b, PTR1 78b, and PTR2 78b.

8.5 Digging

Provided there's nothing preventing the player from digging, digging involves a brick animation below and next to the player, and a "debris" animation above the dig site.



The DIG_DIRECTION location stores which direction we're digging in, and the DIG_ANIM_STATE location stores how far along in the 13-step animation cycle we are.

166 ⟨*defines* 4⟩+≡ (281) ◁144b 181▷

```
DIG_DIRECTION       EQU     $9C     ; 0xFF = left, 0x00 = not digging, 0x01 = right
DIG_ANIM_STATE      EQU     $A0     ; 00-0C
```

Defines:
DIG_DIRECTION, used in chunks 168, 171, 174, 175, and 250.

The DIG_DEBRIS_LEFT_SPRITES, DIG_DEBRIS_RIGHT_SPRITES and DIG_BRICK_SPRITES tables contain the sprites used during the animation. There's also a little sequence of notes that plays while digging, given by DIG_NOTE_PITCHES and DIG_NOTE_DURATIONS.

167 ⟨*tables* 9⟩+≡ (281) ◁150 182▷

```
    ORG     $697A
DIG_DEBRIS_LEFT_SPRITES:
    HEX     1B 1B 1C 1C 1D 1D 1E 1E 00 00 00 00
DIG_DEBRIS_RIGHT_SPRITES:
    HEX     26 26 27 27 1D 1D 1E 1E 00 00 00 00
DIG_BRICK_SPRITES:
    HEX     1F 1F 20 20 21 21 22 22 23 23 24 24
DIG_NOTE_PITCHES:
    HEX     20 20 20 20 20 20 20 20 24 24 24 24 24
DIG_NOTE_DURATIONS:
    HEX     04 04 04 04 04 04 04 04 03 03 02 02 01
```

Defines:
DIG_BRICK_SPRITES, used in chunks 168 and 171.
DIG_DEBRIS_LEFT_SPRITES, used in chunks 168 and 171.
DIG_DEBRIS_RIGHT_SPRITES, never used.
DIG_NOTE_DURATIONS, used in chunks 168 and 171.
DIG_NOTE_PITCHES, used in chunks 168 and 171.

The player cannot dig to the left if they're on the bottom-most row or the leftmost column, or if there's no brick below and to the left. Also, there has to be nothing to the left of the player.

168 ⟨*try digging left* 168⟩≡ (278)

```
    ORG     $67D8
    SUBROUTINE

.cannot_dig_:
    JMP     .stop_digging

TRY_DIGGING_LEFT:
    LDA     #$FF
    STA     DIG_DIRECTION
    STA     KEY_COMMAND
    STA     KEY_COMMAND_LR      ; DIG_DIRECTION = KEY_COMMANDs = 0xFF
    LDA     #$00
    STA     DIG_ANIM_STATE      ; DIG_ANIM_STATE = 0

TRY_DIGGING_LEFT_check_can_dig_left:
    LDY     PLAYER_ROW
    CPY     #MAX_GAME_ROW
    BCS     .cannot_dig_        ; row >= 15, so cannot dig.

    INY
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    BEQ     .cannot_dig_        ; col == 0, so cannot dig left.

    DEY
    LDA     (PTR1),Y
    CMP     #SPRITE_BRICK
    BNE     .cannot_dig_        ; no brick below and to the left, so cannot dig left.

    LDY     PLAYER_ROW
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    DEY
    LDA     (PTR1),Y
    CMP     #SPRITE_EMPTY
    BNE     .not_empty_to_left  ; not empty to the left, so maybe cannot dig left.

    ; Can dig!
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_COLUMN
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_ROW
    LDY     DIG_ANIM_STATE
    LDA     DIG_NOTE_PITCHES,Y
    LDX     DIG_NOTE_DURATIONS,Y
```

```
    JSR     APPEND_NOTE

    LDX     DIG_ANIM_STATE
    LDA     #$00                ; running left
    CPX     #$06
    BCS     .note_0             ; DIG_ANIM_STATE >= 6
    LDA     #$06                ; digging left
.note_0:
    STA     PLAYER_ANIM_STATE
    JSR     DRAW_PLAYER

    LDX     DIG_ANIM_STATE
    CPX     #$0C
    BEQ     .move_player_left
    CPX     #$00
    BEQ     .draw_curr_dig              ; Don't have to erase previous dig debris sprite

    ; Erase the previous dig debris sprite
    LDA     DIG_DEBRIS_LEFT_SPRITES-1,X
    PHA
    LDX     PLAYER_COL
    DEX
    LDY     PLAYER_ROW
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

    LDX     DIG_ANIM_STATE
.draw_curr_dig:
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDX     PLAYER_COL
    DEX
    STX     GAME_COLNUM
    LDY     PLAYER_ROW
    STY     GAME_ROWNUM
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS     ; Draw current dig debris sprite above dig site

    LDX     DIG_ANIM_STATE
    LDA     DIG_BRICK_SPRITES,X
    INC     GAME_ROWNUM
    JSR     DRAW_SPRITE_PAGE1               ; Draw dig brick sprite at dig site

    INC     DIG_ANIM_STATE
    CLC
    RTS

.not_empty_to_left:
```

```
    LDY     PLAYER_ROW
    INY
    STY     GAME_ROWNUM
    LDY     PLAYER_COL
    DEY
    STY     GAME_COLNUM
    LDA     #SPRITE_BRICK
    JSR     DRAW_SPRITE_PAGE1           ; Draw brick below and to the left of player

    LDX     DIG_ANIM_STATE
    BEQ     .stop_digging

    ; Erase previous dig debris sprite
    DEX
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDY     PLAYER_ROW
    LDX     PLAYER_COL
    DEX
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

.stop_digging:
    LDA     #$00
    STA     DIG_DIRECTION
    SEC
    RTS

.move_player_left:
    LDX     PLAYER_COL
    DEX
    JMP     DROP_PLAYER_IN_HOLE
```

Defines:
TRY_DIGGING_LEFT, used in chunk 175.
Uses APPEND_NOTE 59a, DIG_BRICK_SPRITES 167, DIG_DEBRIS_LEFT_SPRITES 167, DIG_DIRECTION 166, DIG_NOTE_DURATIONS 167, DIG_NOTE_PITCHES 167, DRAW_PLAYER 43, DRAW_SPRITE_AT_PIXEL_COORDS 41, DRAW_SPRITE_PAGE1 35, DROP_PLAYER_IN_HOLE 174, ERASE_SPRITE_AT_PIXEL_COORDS 38, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA 79c, GET_SCREEN_COORDS_FOR 31a, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, KEY_COMMAND 138a, KEY_COMMAND_LR 138a, NUDGE_PLAYER_TOWARDS_EXACT_COLUMN 154, NUDGE_PLAYER_TOWARDS_EXACT_ROW 154, PLAYER_ANIM_STATE 84b, PLAYER_COL 80c, PLAYER_ROW 80c, and PTR1 78b.

171 ⟨*try digging right* 171⟩≡ (278)

```
    ORG     $689E
    SUBROUTINE

.cannot_dig_:
    JMP     .stop_digging

TRY_DIGGING_RIGHT:
    LDA     #$01
    STA     DIG_DIRECTION
    STA     KEY_COMMAND
    STA     KEY_COMMAND_LR      ; DIG_DIRECTION = KEY_COMMANDs = 0x01
    LDA     #$0C
    STA     DIG_ANIM_STATE      ; DIG_ANIM_STATE = 0x0C

TRY_DIGGING_RIGHT_check_can_dig_right:
    LDY     PLAYER_ROW
    CPY     #MAX_GAME_ROW
    BCS     .cannot_dig_        ; row >= 15, so cannot dig.

    INY
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    CPY     #MAX_GAME_COL
    BCS     .cannot_dig_        ; col >= 27, so cannot dig right.

    INY
    LDA     (PTR1),Y
    CMP     #SPRITE_BRICK
    BNE     .cannot_dig_        ; no brick below and to the right, so cannot dig right.

    LDY     PLAYER_ROW
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    INY
    LDA     (PTR1),Y
    CMP     #SPRITE_EMPTY
    BNE     .not_empty_to_right  ; not empty to the right, so maybe cannot dig right.

    ; Can dig!
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_COLUMN
    JSR     NUDGE_PLAYER_TOWARDS_EXACT_ROW
    LDY     DIG_ANIM_STATE
    LDA     DIG_NOTE_PITCHES-12,Y
    LDX     DIG_NOTE_DURATIONS-12,Y
    JSR     APPEND_NOTE

    LDX     DIG_ANIM_STATE
```

```
    LDA     #$08                ; running right
    CPX     #$12
    BCS     .note_0             ; DIG_ANIM_STATE >= 0x12
    LDA     #$0E                ; digging right
.note_0:
    STA     PLAYER_ANIM_STATE
    JSR     DRAW_PLAYER

    LDX     DIG_ANIM_STATE
    CPX     #$18
    BEQ     .move_player_right
    CPX     #$0C
    BEQ     .draw_curr_dig              ; Don't have to erase previous dig debris sprite

    ; Erase the previous dig debris sprite
    LDA     DIG_DEBRIS_LEFT_SPRITES-1,X
    PHA
    LDX     PLAYER_COL
    INX
    LDY     PLAYER_ROW
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

    LDX     DIG_ANIM_STATE
.draw_curr_dig:
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDX     PLAYER_COL
    INX
    STX     GAME_COLNUM
    LDY     PLAYER_ROW
    STY     GAME_ROWNUM
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS     ; Draw current dig debris sprite above dig site

    INC     GAME_ROWNUM
    LDX     DIG_ANIM_STATE
    LDA     DIG_BRICK_SPRITES-12,X
    JSR     DRAW_SPRITE_PAGE1               ; Draw dig brick sprite at dig site

    INC     DIG_ANIM_STATE
    CLC
    RTS

.not_empty_to_right:
    LDY     PLAYER_ROW
    INY
    STY     GAME_ROWNUM
```

```
    LDY     PLAYER_COL
    INY
    STY     GAME_COLNUM
    LDA     #SPRITE_BRICK
    JSR     DRAW_SPRITE_PAGE1           ; Draw brick below and to the right of player

    LDX     DIG_ANIM_STATE
    CPX     #$0C
    BEQ     .stop_digging

    ; Erase previous dig debris sprite
    DEX
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDX     PLAYER_COL
    INX
    LDY     PLAYER_ROW
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

.stop_digging:
    LDA     #$00
    STA     DIG_DIRECTION
    SEC
    RTS

.move_player_right:
    LDX     PLAYER_COL
    INX
    JMP     DROP_PLAYER_IN_HOLE
```

Defines:
TRY_DIGGING_RIGHT, used in chunk 175.
Uses APPEND_NOTE 59a, DIG_BRICK_SPRITES 167, DIG_DEBRIS_LEFT_SPRITES 167, DIG_DIRECTION 166, DIG_NOTE_DURATIONS 167, DIG_NOTE_PITCHES 167, DRAW_PLAYER 43, DRAW_SPRITE_AT_PIXEL_COORDS 41, DRAW_SPRITE_PAGE1 35, DROP_PLAYER_IN_HOLE 174, ERASE_SPRITE_AT_PIXEL_COORDS 38, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA 79c, GET_SCREEN_COORDS_FOR 31a, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, KEY_COMMAND 138a, KEY_COMMAND_LR 138a, NUDGE_PLAYER_TOWARDS_EXACT_COLUMN 154, NUDGE_PLAYER_TOWARDS_EXACT_ROW 154, PLAYER_ANIM_STATE 84b, PLAYER_COL 80c, PLAYER_ROW 80c, and PTR1 78b.

174 ⟨*drop player in hole* 174⟩≡ (278)

```
        ORG     $6C39
    DROP_PLAYER_IN_HOLE:
        SUBROUTINE

        LDA     #$00
        STA     DIG_DIRECTION       ; Stop digging

        LDY     PLAYER_ROW
        INY                         ; Move player down

        STX     GAME_COLNUM
        STY     GAME_ROWNUM
        ⟨set active row pointer PTR1 for Y 78c⟩
        LDA     #SPRITE_EMPTY
        LDY     GAME_COLNUM
        STA     (PTR1),Y                ; Set blank sprite at player location in active page
        JSR     DRAW_SPRITE_PAGE1
        LDA     #SPRITE_EMPTY
        JSR     DRAW_SPRITE_PAGE2       ; Draw blank at player location on both graphics pages

        DEC     GAME_ROWNUM
        LDA     #SPRITE_EMPTY
        JSR     DRAW_SPRITE_PAGE1       ; Draw blank at location above player
        INC     GAME_ROWNUM
        LDX     #$FF

    .loop:
        INX
        CPX     #$1E
        BEQ     .end
        LDA     BRICK_FILL_TIMERS,X
        BNE     .loop

        LDA     GAME_ROWNUM
        STA     BRICK_DIG_ROWS,X
        LDA     GAME_COLNUM
        STA     BRICK_DIG_COLS,X
        LDA     #$B4
        STA     BRICK_FILL_TIMERS,X
        SEC

    .end:
        RTS
```

Defines:
DROP_PLAYER_IN_HOLE, used in chunks 168 and 171.
Uses DIG_DIRECTION 166, DRAW_SPRITE_PAGE1 35, DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, GAME_ROWNUM 34a, PLAYER_ROW 80c, and PTR1 78b.

The MOVE_PLAYER routine handle continuation of digging, player falling, and player keyboard input.

175 ⟨*move player* 175⟩≡ (278)

```
      ORG     $64BD
  MOVE_PLAYER:
      SUBROUTINE

      LDA     #$01
      STA     DIDNT_PICK_UP_GOLD   ; Reset DIDNT_PICK_UP_GOLD

      ; If we're digging, see if we can keep digging.
      LDA     DIG_DIRECTION
      BEQ     .not_digging
      BPL     .digging_right
      JMP     TRY_DIGGING_LEFT_check_can_dig_left

  .digging_right:
      JMP     TRY_DIGGING_RIGHT_check_can_dig_right

  .not_digging:
      LDY     PLAYER_ROW
      ⟨set background row pointer PTR2 for Y 78d⟩
      LDY     PLAYER_COL
      LDA     (PTR2),Y
      CMP     #SPRITE_LADDER
      BEQ     .check_for_keyboard_input_      ; ladder at background location?
      CMP     #SPRITE_ROPE
      BNE     .check_if_player_should_fall    ; rope at background location?
      LDA     PLAYER_Y_ADJ
      CMP     #$02
      BEQ     .check_for_keyboard_input_      ; player at exact sprite row?

      ; player is not on exact sprite row, fallthrough.

  .check_if_player_should_fall:
      LDA     PLAYER_Y_ADJ
      CMP     #$02
      BCC     .make_player_fall               ; player slightly above sprite row?

      LDY     PLAYER_ROW
      CPY     #MAX_GAME_ROW
      BEQ     .check_for_keyboard_input_      ; player exactly sprite row 15?

      ; Check the sprite at the player location
      ⟨set active and background row pointers PTR1 and PTR2 for Y+1 80b⟩

      LDY     PLAYER_COL
      LDA     (PTR1),Y
      CMP     #SPRITE_EMPTY
```

```
    BEQ     .make_player_fall
    CMP     #SPRITE_GUARD
    BEQ     .check_for_keyboard_input_
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BEQ     .check_for_keyboard_input_
    CMP     #SPRITE_STONE
    BEQ     .check_for_keyboard_input_
    CMP     #SPRITE_LADDER
    BNE     .make_player_fall

.check_for_keyboard_input_:
    JMP     .check_for_keyboard_input

.make_player_fall:
    LDA     #$00
    STA     $9B                     ; $9B = 0
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

    LDA     #$07                    ; Next anim state: player falling, facing left
    LDX     PLAYER_FACING_DIRECTION
    BMI     .player_facing_left
    LDA     #$0F                    ; Next anim state: player falling, facing right
.player_facing_left:
    STA     PLAYER_ANIM_STATE

    JSR     NUDGE_PLAYER_TOWARDS_EXACT_COLUMN

    INC     PLAYER_Y_ADJ            ; Move down one
    LDA     PLAYER_Y_ADJ
    CMP     #$05
    BCS     .adjustment_overflow

    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER
    JMP     DRAW_PLAYER            ; tailcall

.adjustment_overflow:
    LDA     #$00
    STA     PLAYER_Y_ADJ            ; Set vertical adjust to -2

    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BNE     .set_on_level
    LDA     #SPRITE_EMPTY
.set_on_level:
    STA     (PTR1),Y
```

```
    INC     PLAYER_ROW          ; Move down
    LDY     PLAYER_ROW
    ⟨set active row pointer PTR1 for Y 78c⟩

    LDY     PLAYER_COL
    LDA     #SPRITE_PLAYER
    STA     (PTR1),Y
    JMP     DRAW_PLAYER     ; tailcall

.check_for_keyboard_input:
    LDA     $9B
    BNE     .check_for_key  ; $9B doesn't play note
    LDA     #$64
    LDX     #$08
    JSR     PLAY_NOTE       ; play note, pitch 0x64, duration 8.

.check_for_key:
    LDA     #$20
    STA     $A4
    STA     $9B
    JSR     CHECK_FOR_INPUT
    LDA     KEY_COMMAND
    CMP     #$C9            ; 'I'
    BNE     .check_for_K
    JSR     TRY_MOVING_UP
    BCS     .check_for_J    ; couldn't move up
    RTS

.check_for_K:
    CMP     #$CB            ; 'K'
    BNE     .check_for_U
    JSR     TRY_MOVING_DOWN
    BCS     .check_for_J
    RTS

.check_for_U:
    CMP     #$D5            ; 'U'
    BNE     .check_for_O
    JSR     TRY_DIGGING_LEFT
    BCS     .check_for_J
    RTS

.check_for_O:
    CMP     #$CF            ; 'O'
    BNE     .check_for_J
    JSR     TRY_DIGGING_RIGHT
    BCS     .check_for_J
    RTS

.check_for_J:
```

```
    LDA     KEY_COMMAND_LR
    CMP     #$CA            ; 'J'
    BNE     .check_for_L
    JMP     TRY_MOVING_LEFT

.check_for_L:
    CMP     #$CC            ; 'L'
    BNE     .end
    JMP     TRY_MOVING_RIGHT

.end:
    RTS
```

Defines:
MOVE_PLAYER, used in chunk 250.
Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 137, CHECK_FOR_INPUT 139, DIDNT_PICK_UP_GOLD 136b, DIG_DIRECTION 166, DRAW_PLAYER 43, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 135b, KEY_COMMAND 138a, KEY_COMMAND_LR 138a, NUDGE_PLAYER_TOWARDS_EXACT_COLUMN 154, PLAY_NOTE 60a, PLAYER_ANIM_STATE 84b, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_Y_ADJ 84b, PTR1 78b, PTR2 78b, TRY_DIGGING_LEFT 168, TRY_DIGGING_RIGHT 171, TRY_MOVING_DOWN 159, TRY_MOVING_LEFT 161, TRY_MOVING_RIGHT 164, and TRY_MOVING_UP 157.

ENABLE_NEXT_LEVEL_LADDERS goes through the registered ladder locations from last to first. Recall that the ladder indices are 1-based, so that LADDER_LOCS_[0] does not contain ladder data. Instead, that location is used as scratch space by this routine.

Recall also that LADDER_LOCS_[X] is negative if there is no ladder corresponding to entry X.

For each ladder, if there's a non-blank sprite on the background sprite page for it, we set LADDER_LOCS_COL to 1.

However, if there is a blank sprite on the background sprite page for it, then set it to the ladder sprite, and if it's also blank on the active sprite page, set that to the ladder sprite, too. Then draw the ladder on the background and active graphics pages, remove the ladder from the registered locations, and keep going.

Once all ladder locations have been gone through, if LADDER_LOCS_COL is 1—that is, if there was a non-blank sprite on the background sprite page for any ladder location—then decrement the gold count. Since this routine is only called when GOLD_COUNT is zero, this sets GOLD_COUNT to -1.

179 ⟨*do ladders* 179⟩≡ (278)

```
    ORG     $8631
ENABLE_NEXT_LEVEL_LADDERS:
    SUBROUTINE

    LDA     #$00
    STA     LADDER_LOCS_COL     ; LADDER_LOCS_COL = 0

    LDX     LADDER_COUNT
    STX     .count              ; .count backwards from LADDER_COUNT to 0
.loop:
    LDX     .count
    BEQ     .dec_gold_count_if_no_ladder

    LDA     LADDER_LOCS_COL,X   ; A = LADDER_LOCS_COL[X]
    BMI     .next               ; If not present, next.

    STA     GAME_COLNUM         ; GAME_COLNUM = LADDER_LOCS_COL[X]
    LDA     LADDER_LOCS_ROW,X
    STA     GAME_ROWNUM         ; GAME_ROWNUM = LADDER_LOCS_ROW[X]
    TAY
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDY     GAME_COLNUM
    LDA     (PTR2),Y            ; A = sprite at ladder loc
    BNE     .set_col_to_1

    LDA     #SPRITE_LADDER
    STA     (PTR2),Y            ; Set background sprite to ladder
    LDA     (PTR1),Y
    BNE     .draw_ladder        ; .draw_ladder if active sprite not blank
```

```
    LDA     #SPRITE_LADDER
    STA     (PTR1),Y            ; Set active sprite to ladder

.draw_ladder:
    LDA     #SPRITE_LADDER
    JSR     DRAW_SPRITE_PAGE2   ; Draw ladder on background page

    LDX     GAME_COLNUM
    LDY     GAME_ROWNUM
    JSR     GET_SCREEN_COORDS_FOR
    LDA     #SPRITE_LADDER
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS ; Draw ladder on active page

    LDX     .count
    LDA     #$FF
    STA     LADDER_LOCS_COL,X       ; Remove ladder loc
    BMI     .next                   ; Unconditional

.set_col_to_1:
    LDA     #$01
    STA     LADDER_LOCS_COL         ; LADDER_LOCS_COL = 1

.next:
    DEC     .count
    JMP     .loop

.dec_gold_count_if_no_ladder:
    LDA     LADDER_LOCS_COL
    BNE     .end
    DEC     GOLD_COUNT

.end:
    RTS

.count:
    BYTE    0
```

Defines:
ENABLE_NEXT_LEVEL_LADDERS, used in chunk 250.
Uses DRAW_SPRITE_AT_PIXEL_COORDS 41, DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_SCREEN_COORDS_FOR 31a, GOLD_COUNT 81d, LADDER_COUNT 81d, LADDER_LOCS_COL 82a, LADDER_LOCS_ROW 82a, PTR1 78b, and PTR2 78b.

Chapter 9

Guard AI

Like the player, each guard has a column and row sprite location and a horizontal and vertical adjustment. Each guard also has an animation state and a facing direction.

Guards also maintain two timers: a gold timer and a resurrection timer. The resurrection timer comes into play when a guard is killed by a closing hole.

181 ⟨*defines* 4⟩+≡ (281) ◁166 190▷

```
GUARD_LOCS_COL      EQU     $0C60       ; 8 bytes
GUARD_LOCS_ROW      EQU     $0C68       ; 8 bytes
GUARD_GOLD_TIMERS   EQU     $0C70       ; 8 bytes
GUARD_X_ADJS        EQU     $0C78       ; 8 bytes
GUARD_Y_ADJS        EQU     $0C80       ; 8 bytes
GUARD_ANIM_STATES   EQU     $0C88       ; 8 bytes
GUARD_FACING_DIRECTIONS     EQU     $0C90       ; 8 bytes
GUARD_RESURRECTION_TIMERS   EQU     $0C98       ; 8 bytes

GUARD_LOC_COL       EQU     $12
GUARD_LOC_ROW       EQU     $13
GUARD_ANIM_STATE    EQU     $14
GUARD_FACING_DIRECTION      EQU     $15     ; Hi bit set: facing left, otherwise facing right
GUARD_GOLD_TIMER    EQU     $16
GUARD_X_ADJ         EQU     $17
GUARD_Y_ADJ         EQU     $18
GUARD_NUM           EQU     $19

GUARD_PATTERN       EQU     $63
GUARD_PHASE         EQU     $64

GUARD_RESURRECT_COL EQU     $53
TMP_GUARD_COL       EQU     $55
TMP_GUARD_ROW       EQU     $56
GUARD_GOLD_TIMER_START_VALUE        EQU     $5F
```

Defines:
GUARD_ANIM_STATE, used in chunks 185–87 and 193.

GUARD_ANIM_STATES, used in chunks 83b, 126, and 186.
GUARD_FACING_DIRECTION, used in chunks 186, 193, 204, and 206.
GUARD_FACING_DIRECTIONS, used in chunk 186.
GUARD_GOLD_TIMER, used in chunks 184, 186, 193, 198, 208, and 212.
GUARD_GOLD_TIMER_START_VALUE, used in chunks 193 and 250.
GUARD_GOLD_TIMERS, used in chunks 83b, 126, 186, and 188.
GUARD_LOC_COL, used in chunks 184, 186, 187b, 193, 204, 206, 208, 210, 212, and 213.
GUARD_LOC_ROW, used in chunks 184, 186, 187b, 193, 204, 206, 208, 210, and 212.
GUARD_LOCS_COL, used in chunks 83b, 126, 145, 186, and 188.
GUARD_LOCS_ROW, used in chunks 83b, 126, 145, 186, and 188.
GUARD_NUM, used in chunks 112b, 126, 186, 188, and 193.
GUARD_PATTERN, used in chunk 191a.
GUARD_PHASE, used in chunk 191a.
GUARD_X_ADJ, used in chunks 183, 184, 186, 187b, 193, 204, and 206.
GUARD_X_ADJS, used in chunks 83b, 126, and 186.
GUARD_Y_ADJ, used in chunks 183, 184, 186, 187b, 193, 208, and 210.
GUARD_Y_ADJS, used in chunks 83b, 126, and 186.

182 ⟨*tables* 9⟩+≡ (281) ◁167 187a▷

```
    ORG     $621D
GUARD_GOLD_TIMER_START_VALUES:
    HEX     26 26 2E 44 47 49 4A 4B 4C 4D 4E 4F 50
```

Defines:
GUARD_GOLD_TIMER_START_VALUES, never used.

183 ⟨*nudge guards* 183⟩≡ (278)

```
      ORG     $7582
  NUDGE_GUARD_TOWARDS_EXACT_COLUMN:
      SUBROUTINE

      LDA     GUARD_X_ADJ
      CMP     #$02
      BCC     .slightly_left
      BEQ     .end

  .slightly_right:
      DEC     GUARD_X_ADJ        ; Nudge guard left
      JMP     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD

  .slightly_left:
      INC     GUARD_X_ADJ        ; Nudge guard right
      JMP     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD

  .end:
      RTS

      ORG     $7595
  NUDGE_GUARD_TOWARDS_EXACT_ROW:
      SUBROUTINE

      LDA     GUARD_Y_ADJ
      CMP     #$02
      BCC     .slightly_above
      BEQ     .end

  .slightly_below:
      DEC     GUARD_Y_ADJ        ; Nudge guard up
      JMP     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD

  .slightly_above:
      INC     GUARD_Y_ADJ        ; Nudge guard down
      JMP     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD

  .end:
      RTS
```

Defines:
NUDGE_GUARD_TOWARDS_EXACT_COLUMN, used in chunks 193, 208, and 210.
NUDGE_GUARD_TOWARDS_EXACT_ROW, used in chunks 204 and 206.
Uses CHECK_FOR_GOLD_PICKED_UP_BY_GUARD 184, GUARD_X_ADJ 181, and GUARD_Y_ADJ 181.

If the guard is exactly on a sprite coordinate, and there's gold there, and GUARD_GOLD_TIMER is zero or positive, then set GUARD_GOLD_TIMER to 0xFF - $53, and remove the gold.

184 ⟨*check for gold picked up by guard* 184⟩≡ (278)

```
    ORG     $74F7
CHECK_FOR_GOLD_PICKED_UP_BY_GUARD:
    SUBROUTINE

    LDA     GUARD_X_ADJ
    CMP     #$02
    BNE     .end
    LDA     GUARD_Y_ADJ
    CMP     #$02
    BNE     .end

    LDY     GUARD_LOC_ROW
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y

    CMP     #SPRITE_GOLD
    BNE     .end

    LDA     GUARD_GOLD_TIMER        ; Does guard have gold already?
    BMI     .end

    LDA     #$FF
    SEC
    SBC     $53
    STA     GUARD_GOLD_TIMER        ; GUARD_GOLD_TIMER = 0xFF - $53

    ; Remove gold from screen
    LDA     #SPRITE_EMPTY
    STA     (PTR2),Y
    LDY     GUARD_LOC_ROW
    STY     GAME_ROWNUM
    LDY     GUARD_LOC_COL
    STY     GAME_COLNUM
    JSR     DRAW_SPRITE_PAGE2

    LDY     GAME_ROWNUM
    LDX     GAME_COLNUM
    JSR     GET_SCREEN_COORDS_FOR
    LDA     #SPRITE_GOLD
    JMP     ERASE_SPRITE_AT_PIXEL_COORDS        ; tailcall

.end:
    RTS
```

Defines:

CHECK_FOR_GOLD_PICKED_UP_BY_GUARD, used in chunks 183, 193, 204, 206, and 208.
Uses DRAW_SPRITE_PAGE2 35, ERASE_SPRITE_AT_PIXEL_COORDS 38, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_SCREEN_COORDS_FOR 31a, GUARD_GOLD_TIMER 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_X_ADJ 181, GUARD_Y_ADJ 181, and PTR2 78b.

185 ⟨*increment guard animation state* 185⟩≡ (278)

```
    ORG     $7574
INC_GUARD_ANIM_STATE:
    SUBROUTINE

    INC     GUARD_ANIM_STATE
    CMP     GUARD_ANIM_STATE
    BCC     .check_upper_bound      ; lower bound < GUARD_ANIM_STATE?
    ; otherwise PLAYER_ANIM_STATE <= lower bound:

.write_lower_bound:
    STA     GUARD_ANIM_STATE       ; GUARD_ANIM_STATE = lower bound
    RTS

.check_upper_bound:
    CPX     GUARD_ANIM_STATE
    BCC     .write_lower_bound       ; GUARD_ANIM_STATE > upper bound?
    ; otherwise GUARD_ANIM_STATE <= upper bound:
    RTS
```

Defines:
INC_GUARD_ANIM_STATE, used in chunks 204, 206, and 208.
Uses GUARD_ANIM_STATE 181 and PLAYER_ANIM_STATE 84b.

186 ⟨*guard store and load data* 186⟩≡ (278)

```
      ORG     $75A8
  STORE_GUARD_DATA:
      SUBROUTINE

      LDX     GUARD_NUM
      LDA     GUARD_LOC_COL
      STA     GUARD_LOCS_COL,X
      LDA     GUARD_LOC_ROW
      STA     GUARD_LOCS_ROW,X
      LDA     GUARD_X_ADJ
      STA     GUARD_X_ADJS,X
      LDA     GUARD_Y_ADJ
      STA     GUARD_Y_ADJS,X
      LDA     GUARD_GOLD_TIMER
      STA     GUARD_GOLD_TIMERS,X
      LDA     GUARD_FACING_DIRECTION
      STA     GUARD_FACING_DIRECTIONS,X
      LDA     GUARD_ANIM_STATE
      STA     GUARD_ANIM_STATES,X
      RTS

  LOAD_GUARD_DATA:
      SUBROUTINE

      LDX     GUARD_NUM
      LDA     GUARD_LOCS_COL,X
      STA     GUARD_LOC_COL
      LDA     GUARD_LOCS_ROW,X
      STA     GUARD_LOC_ROW
      LDA     GUARD_X_ADJS,X
      STA     GUARD_X_ADJ
      LDA     GUARD_Y_ADJS,X
      STA     GUARD_Y_ADJ
      LDA     GUARD_ANIM_STATES,X
      STA     GUARD_ANIM_STATE
      LDA     GUARD_FACING_DIRECTIONS,X
      STA     GUARD_FACING_DIRECTION
      LDA     GUARD_GOLD_TIMERS,X
      STA     GUARD_GOLD_TIMER
      RTS
```

Defines:
  LOAD_GUARD_DATA, used in chunks 126, 188, and 193.
  STORE_GUARD_DATA, used in chunks 192b, 193, 204, 206, 208, and 210.
Uses GUARD_ANIM_STATE 181, GUARD_ANIM_STATES 181, GUARD_FACING_DIRECTION 181, GUARD_FACING_DIRECTIONS 181, GUARD_GOLD_TIMER 181, GUARD_GOLD_TIMERS 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_LOCS_COL 181, GUARD_LOCS_ROW 181, GUARD_NUM 181, GUARD_X_ADJ 181, GUARD_X_ADJS 181, GUARD_Y_ADJ 181, and GUARD_Y_ADJS 181.

187a ⟨*tables* 9⟩+≡ (281) ◁182 191b▷

```
      ORG     $6CCB
  GUARD_ANIM_SPRITES:
      HEX     08 2B 2C        ; running left
      HEX     30 31 32        ; monkey-traversing left
      HEX     36              ; falling left
      HEX     28 29 2A        ; running right
      HEX     2D 2E 2F        ; monkey-traversing right
      HEX     35              ; falling right
      HEX     33 34           ; climbing
```

Defines:
GUARD_ANIM_SPRITES, used in chunk 187b.

187b ⟨*get guard sprite and coords* 187b⟩≡ (278)

```
      ORG     $74DF
  GET_GUARD_SPRITE_AND_COORDS:
      SUBROUTINE

      LDX     GUARD_LOC_COL
      LDY     GUARD_X_ADJ
      JSR     GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR
      STX     SPRITE_NUM

      LDY     GUARD_LOC_ROW
      LDX     GUARD_Y_ADJ
      JSR     GET_SCREEN_ROW_OFFSET_IN_X_FOR
      LDX     GUARD_ANIM_STATE
      LDA     GUARD_ANIM_SPRITES,X

      LDX     SPRITE_NUM
      RTS
```

Defines:
GET_GUARD_SPRITE_AND_COORDS, used in chunks 126, 188, 193, 204, 206, 208, and 210.
Uses GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR 33c, GET_SCREEN_ROW_OFFSET_IN_X_FOR 33a, GUARD_ANIM_SPRITES 187a, GUARD_ANIM_STATE 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_X_ADJ 181, GUARD_Y_ADJ 181, and SPRITE_NUM 25c.

The GUARD_RESURRECTIONS routine handles guard resurrection. It checks each guard's resurrection timer to see if it is nonzero. If so, we decrement the guard's timer and then check the timer for specific values:

- 19: Draw the #SPRITE_GUARD_EGG0 sprite at the guard's location.
- 10: Draw the #SPRITE_GUARD_EGG1 sprite at the guard's location.
- 0: Increment the timer and check if the guard's location is empty on the active page. If so, put the #SPRITE_GUARD sprite at the guard's location, set its timers to zero, and play the guard resurrection sound.

188 ⟨*guard resurrections* 188⟩≡ (278)

```
    ORG     $7716
GUARD_RESURRECTIONS:
    SUBROUTINE

    LDX     GUARD_COUNT
    BEQ     HANDLE_TIMERS_COMMON_RETURN

    LDA     GUARD_NUM
    PHA

.loop:
    LDA     GUARD_RESURRECTION_TIMERS,X
    BEQ     .next

    STX     GUARD_NUM
    JSR     LOAD_GUARD_DATA
    LDA     #$7F
    STA     GUARD_GOLD_TIMERS,X
    LDA     GUARD_LOCS_COL,X
    STA     GAME_COLNUM
    LDA     GUARD_LOCS_ROW,X
    STA     GAME_ROWNUM

    DEC     GUARD_RESURRECTION_TIMERS,X
    BEQ     .resurrect

    LDA     GUARD_RESURRECTION_TIMERS,X
    CMP     #$13                ; 19
    BNE     .check_guard_flag_5_is_10

    ; GUARD_RESURRECTION_TIMER is 19

    LDA     #SPRITE_GUARD_EGG0
    JSR     DRAW_SPRITE_PAGE2
    JSR     GET_GUARD_SPRITE_AND_COORDS
    LDA     #SPRITE_GUARD_EGG0
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
```

```
    JMP     .next2

.check_guard_flag_5_is_10:
    CMP     #$0A                ; 10
    BNE     .next

    ; GUARD_RESURRECTION_TIMER is 10
    LDA     #SPRITE_GUARD_EGG1
    JSR     DRAW_SPRITE_PAGE2
    JSR     GET_GUARD_SPRITE_AND_COORDS
    LDA     #SPRITE_GUARD_EGG1
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS

.next2:
    ; Restores the counter
    LDX     GUARD_NUM

.next:
    DEX
    BNE     .loop

    PLA
    STA     GUARD_NUM
    RTS

.resurrect:
    LDY     GAME_ROWNUM
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDX     GUARD_NUM
    INC     GUARD_RESURRECTION_TIMERS,X
    LDY     GAME_COLNUM
    LDA     (PTR1),Y
    BNE     .next

    ; empty
    LDA     #SPRITE_GUARD
    STA     (PTR1),Y
    LDA     #SPRITE_EMPTY
    JSR     DRAW_SPRITE_PAGE2
    LDA     #$00
    LDX     GUARD_NUM
    STA     GUARD_GOLD_TIMERS,X
    STA     GUARD_RESURRECTION_TIMERS,X
    LDA     #SPRITE_GUARD
    JSR     DRAW_SPRITE_PAGE1

    ; Play the "guard resurrection" sound
    JSR     LOAD_SOUND_DATA
    HEX     02 7C 03 78 04 74 05 70 00
```

```
      LDX     GUARD_NUM
      JMP     .next
```

Uses DRAW_SPRITE_AT_PIXEL_COORDS 41, DRAW_SPRITE_PAGE1 35, DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_GUARD_SPRITE_AND_COORDS 187b, GUARD_COUNT 81d, GUARD_GOLD_TIMERS 181, GUARD_LOCS_COL 181, GUARD_LOCS_ROW 181, GUARD_NUM 181, LOAD_GUARD_DATA 186, LOAD_SOUND_DATA 58, and PTR1 78b.

190 ⟨*defines* 4⟩+≡ (281) ◁181 192a▷

```
      ; Init with:
      ; BYTE    %10000110
      ; BYTE    %00111110
      ; BYTE    %10000101
GUARD_PATTERNS  EQU     $60     ; 3 bytes
```

Defines:
GUARD_PATTERNS, used in chunks 191a and 250.

191a ⟨*move guards* 191a⟩≡ (278)

```
        ORG     $6C82
    MOVE_GUARDS:
        SUBROUTINE

        LDX     GUARD_COUNT
        BEQ     .end

        ; Increment GUARD_PHASE mod 3
        INC     GUARD_PHASE
        LDY     GUARD_PHASE
        CPY     #$03
        BCC     .incremented_phase
        LDY     #$00
        STY     GUARD_PHASE
    .incremented_phase:

        LDA     GUARD_PATTERNS,Y
        STA     GUARD_PATTERN

    .loop:
        LSR     GUARD_PATTERN       ; Peel off the lsb
        BCC     .bit_done
        JSR     MOVE_GUARD          ; Move a guard
        LDA     ALIVE
        BEQ     .end                ; If player is dead, end.

    .bit_done:
        LDA     GUARD_PATTERN
        BNE     .loop

    .end:
        RTS
```

Defines:
MOVE_GUARDS, used in chunk 250.
Uses ALIVE 111a, GUARD_COUNT 81d, GUARD_PATTERN 181, GUARD_PATTERNS 190, GUARD_PHASE 181, and MOVE_GUARD 193.

191b ⟨*tables* 9⟩+≡ (281) ◁187a 192b▷

```
        ORG     $6E7F
    GUARD_X_ADJ_TABLE:
        HEX     02 01 02 03 02 01
```

Defines:
GUARD_X_ADJ_TABLE, used in chunk 193.

192a ⟨*defines* 4⟩+≡ (281) ◁190 197▷

```
GUARD_ACTION    EQU     $58     ; Index into GUARD_FN_TABLE

GUARD_ACTION_DO_NOTHING     EQU     #$00
GUARD_ACTION_MOVE_LEFT      EQU     #$01
GUARD_ACTION_MOVE_RIGHT     EQU     #$02
GUARD_ACTION_MOVE_UP        EQU     #$03
GUARD_ACTION_MOVE_DOWN      EQU     #$04
```

Defines:
GUARD_ACTION, used in chunks 198, 215, 217, and 219.
Uses GUARD_FN_TABLE 192b.

192b ⟨*tables* 9⟩+≡ (281) ◁191b 230▷

```
    ORG     $6E97
GUARD_FN_TABLE:
    WORD    STORE_GUARD_DATA-1
    WORD    TRY_GUARD_MOVE_LEFT-1
    WORD    TRY_GUARD_MOVE_RIGHT-1
    WORD    TRY_GUARD_MOVE_UP-1
    WORD    TRY_GUARD_MOVE_DOWN-1
```

Defines:
GUARD_FN_TABLE, used in chunks 192a and 193.
Uses STORE_GUARD_DATA 186, TRY_GUARD_MOVE_DOWN 210, TRY_GUARD_MOVE_LEFT 204, TRY_GUARD_MOVE_RIGHT 206, and TRY_GUARD_MOVE_UP 208.

193 ⟨*move guard* 193⟩≡ (278)

```
        ORG     $6CDB
    MOVE_GUARD
        SUBROUTINE

        ; Increment GUARD_NUM mod GUARD_COUNT, except 1-based.
        INC     GUARD_NUM
        LDX     GUARD_COUNT
        CPX     GUARD_NUM
        BCS     .guard_num_incremented
        LDX     #$01
        STX     GUARD_NUM
    .guard_num_incremented:

        JSR     LOAD_GUARD_DATA
        LDA     GUARD_GOLD_TIMER
        BMI     .check_sprite_at_guard_pos
        BEQ     .check_sprite_at_guard_pos

        ; GUARD_GOLD_TIMER > 0:
        DEC     GUARD_GOLD_TIMER
        LDY     GUARD_GOLD_TIMER
        CPY     #$0D
        BCS     .guard_gold_timer_expired

        ; GUARD_GOLD_TIMER < 12
        JMP     .check_gold_timer

    .guard_gold_timer_expired:
        LDX     GUARD_NUM
        LDA     GUARD_RESURRECTION_TIMERS,X
        BEQ     .resurrect_guard_
        JMP     STORE_GUARD_DATA            ; tailcall

    .resurrect_guard_:
        JMP     .resurrect_guard

    .check_sprite_at_guard_pos:
        LDY     GUARD_LOC_ROW
        ⟨set background row pointer PTR2 for Y 78d⟩
        LDY     GUARD_LOC_COL
        LDA     (PTR2),Y

        CMP     #SPRITE_LADDER
        BEQ     .ladder_
        CMP     #SPRITE_ROPE
        BNE     .not_rope_or_ladder
        LDA     GUARD_Y_ADJ
        CMP     #$02
        BEQ     .ladder_
```

```
.not_rope_or_ladder:
    LDA     GUARD_Y_ADJ
    CMP     #$02
    BCC     .blank_or_player           ; if GUARD_Y_ADJ < 2
    LDY     GUARD_LOC_ROW
    CPY     #MAX_GAME_ROW
    BEQ     .ladder_           ; Row == 15
    ⟨set active and background row pointers PTR2 and PTR1 for Y+1 79b⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR1),Y

    CMP     #SPRITE_EMPTY
    BEQ     .blank_or_player
    CMP     #SPRITE_PLAYER
    BEQ     .blank_or_player
    CMP     #SPRITE_GUARD
    BEQ     .ladder_

    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BEQ     .ladder_
    CMP     #SPRITE_STONE
    BEQ     .ladder_
    CMP     #SPRITE_LADDER
    BNE     .blank_or_player

.ladder_:
    JMP     .ladder

.blank_or_player:
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    JSR     NUDGE_GUARD_TOWARDS_EXACT_COLUMN
    LDA     #$06
    LDY     GUARD_FACING_DIRECTION
    BMI     .set_guard_anim_state
    LDA     #$0D
.set_guard_anim_state
    STA     GUARD_ANIM_STATE

    INC     GUARD_Y_ADJ
    LDA     GUARD_Y_ADJ
    CMP     #$05
    BCS     $6dc0           ; If GUARD_Y_ADJ > 4

    LDA     GUARD_Y_ADJ
    CMP     #$02
    BNE     .resurrect_guard
```

```
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD
    LDY     GUARD_LOC_ROW
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y

    CMP     #SPRITE_BRICK
    BNE     .resurrect_guard

    LDA     GUARD_GOLD_TIMER
    BPL     .reset_gold_timer
    DEC     GOLD_COUNT
.reset_gold_timer:
    LDA     GUARD_GOLD_TIMER_START_VALUE
    STA     GUARD_GOLD_TIMER
    LDY     #$00
    LDA     #$75
    JSR     ADD_AND_UPDATE_SCORE        ; SCORE += 75

    ; Play the guard kill tune
    JSR     LOAD_SOUND_DATA
    HEX     06 20 04 30 02 40 00

.resurrect_guard:
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA            ; tailcall

.6dc0:
    LDA     #$00
    STA     GUARD_Y_ADJ                 ; set vertical adjust to -2
    LDY     GUARD_LOC_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BNE     .set_real_sprite
    LDA     #SPRITE_EMPTY
.set_real_sprite:
    STA     (PTR1),Y

    INC     GUARD_LOC_ROW               ; move guard down
    LDY     GUARD_LOC_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR1),Y
    CMP     #SPRITE_PLAYER
    BNE     .get_background_sprite
    LSR     ALIVE                       ; set player to dead
.get_background_sprite:
```

```
    LDA     (PTR2),Y
    CMP     #SPRITE_BRICK
    BNE     .place_guard_at_loc
    LDA     GUARD_GOLD_TIMER
    BPL     .place_guard_at_loc

    ; What's above the guard?
    LDY     GUARD_LOC_ROW
    DEY
    STY     GAME_ROWNUM
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    LDY     GUARD_LOC_COL
    STY     GAME_COLNUM
    LDA     (PTR2),Y
    CMP     #SPRITE_EMPTY
    BEQ     .drop_gold
    DEC     GOLD_COUNT
    JMP     .6e46

.drop_gold:
    LDA     #SPRITE_GOLD
    STA     (PTR1),Y
    STA     (PTR2),Y
    JSR     DRAW_SPRITE_PAGE2
    LDY     GAME_ROWNUM
    LDX     GAME_COLNUM
    JSR     GET_SCREEN_COORDS_FOR
    LDA     #SPRITE_GOLD
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS

.6e46
    LDY     GUARD_LOC_ROW
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDA     #$00
    STA     GUARD_GOLD_TIMER
    LDY     GUARD_LOC_COL

.place_guard_at_loc
    LDA     #SPRITE_GUARD
    STA     (PTR1),Y

    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA            ; tailcall

.check_gold_timer:
    CPY     #$07
    BCC     .ladder

    JSR     GET_GUARD_SPRITE_AND_COORDS
```

```
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDY     GUARD_GOLD_TIMER
    LDA     GUARD_X_ADJ_TABLE-7,Y       ; GUARD_X_ADJ_TABLE[GUARD_GOLD_TIMER-7]
    STA     GUARD_X_ADJ
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA            ; tailcall

    ORG     $6E85
.ladder
    LDX     GUARD_LOC_COL
    LDY     GUARD_LOC_ROW
    JSR     DETERMINE_GUARD_MOVE

    ; Go to a guard movement function in the GUARD_FN_TABLE
    ASL
    TAY
    LDA     GUARD_FN_TABLE+1,Y
    PHA
    LDA     GUARD_FN_TABLE,Y
    PHA
    RTS
```

Defines:
MOVE_GUARD, used in chunk 191a.
Uses ADD_AND_UPDATE_SCORE 51, ALIVE 111a, CHECK_FOR_GOLD_PICKED_UP_BY_GUARD 184, DETERMINE_GUARD_MOVE 198, DRAW_SPRITE_AT_PIXEL_COORDS 41, DRAW_SPRITE_PAGE2 35, ERASE_SPRITE_AT_PIXEL_COORDS 38, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_GUARD_SPRITE_AND_COORDS 187b, GET_SCREEN_COORDS_FOR 31a, GOLD_COUNT 81d, GUARD_ANIM_STATE 181, GUARD_COUNT 81d, GUARD_FACING_DIRECTION 181, GUARD_FN_TABLE 192b, GUARD_GOLD_TIMER 181, GUARD_GOLD_TIMER_START_VALUE 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_NUM 181, GUARD_X_ADJ 181, GUARD_X_ADJ_TABLE 191b, GUARD_Y_ADJ 181, LOAD_GUARD_DATA 186, LOAD_SOUND_DATA 58, NUDGE_GUARD_TOWARDS_EXACT_COLUMN 183, PTR1 78b, PTR2 78b, SCORE 50b, and STORE_GUARD_DATA 186.

To determine which direction a guard should move, all directions are checked for a candidate target, and the minimum distance from the candidate target to the player determines the move.

197 ⟨*defines* 4⟩+≡ (281) ◁192a 200▷

```
BEST_GUARD_DIST     EQU     $59
```

Defines:
BEST_GUARD_DIST, used in chunks 198, 215, 217, and 219.

198 ⟨*determine guard move* 198⟩≡ (278)

```
      ORG     $70D8
  DETERMINE_GUARD_MOVE:
      SUBROUTINE

      STX     TMP_GUARD_COL
      STY     TMP_GUARD_ROW
      ⟨set background row pointer PTR2 for Y 78d⟩
      LDY     TMP_GUARD_COL
      LDA     (PTR2),Y

      CMP     #SPRITE_BRICK
      BNE     .end_if_row_is_not_player_row
      LDA     GUARD_GOLD_TIMER
      BEQ     .end_if_row_is_not_player_row
      BMI     .end_if_row_is_not_player_row
      LDA     #GUARD_ACTION_MOVE_UP
      RTS

  .end_if_row_is_not_player_row:
      LDY     TMP_GUARD_ROW
      CPY     PLAYER_ROW
      BEQ     .7100
      JMP     .end

  .7100:
      LDY     TMP_GUARD_COL
      STY     $57
      CPY     PLAYER_COL
      BCS     .loop2
      ; If TMP_GUARD_COL < PLAYER_COL:

  .loop:
      INC     $57
      LDY     TMP_GUARD_ROW
      ⟨set background row pointer PTR2 for Y 78d⟩
      LDY     $57
      LDA     (PTR2),Y

      CMP     #SPRITE_LADDER
      BEQ     .is_ladder_or_rope
      CMP     #SPRITE_ROPE
      BEQ     .is_ladder_or_rope

      LDY     TMP_GUARD_ROW
      CPY     #MAX_GAME_ROW
      BEQ     .is_ladder_or_rope

      ⟨set background row pointer PTR2 for Y+1 80a⟩
      LDY     $57
```

```
    LDA     (PTR2),Y
    CMP     #SPRITE_EMPTY
    BEQ     .end
    CMP     #SPRITE_TRAP
    BEQ     .end

.is_ladder_or_rope:
    LDY     $57
    CPY     PLAYER_COL
    BNE     .loop

    ; PLAYER_COL == $57:
    LDA     #GUARD_ACTION_MOVE_RIGHT
    RTS

.loop2:
    DEC     $57
    LDY     TMP_GUARD_ROW
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     $57
    LDA     (PTR2),Y

    CMP     #SPRITE_LADDER
    BEQ     .is_ladder_or_rope2
    CMP     #SPRITE_ROPE
    BEQ     .is_ladder_or_rope2

    LDY     TMP_GUARD_ROW
    CPY     #MAX_GAME_ROW
    BEQ     .is_ladder_or_rope2

    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     $57
    LDA     (PTR2),Y
    CMP     #SPRITE_EMPTY
    BEQ     .end
    CMP     #SPRITE_TRAP
    BEQ     .end

.is_ladder_or_rope2:
    LDY     $57
    CPY     PLAYER_COL
    BNE     .loop2

    ; PLAYER_COL == $57:
    LDA     #GUARD_ACTION_MOVE_LEFT
    RTS

.end:
    LDA     #GUARD_ACTION_DO_NOTHING
```

```
    STA     GUARD_ACTION
    LDA     #$FF
    STA     BEST_GUARD_DIST
    LDX     TMP_GUARD_COL
    LDY     TMP_GUARD_ROW
    JSR     DETERMINE_GUARD_LEFT_RIGHT_LIMITS
    JSR     SHOULD_GUARD_MOVE_UP_OR_DOWN
    JSR     SHOULD_GUARD_MOVE_LEFT
    JSR     SHOULD_GUARD_MOVE_RIGHT
    LDA     GUARD_ACTION
    RTS
```

Defines:
DETERMINE_GUARD_MOVE, used in chunk 193.
Uses BEST_GUARD_DIST 197, DETERMINE_GUARD_LEFT_RIGHT_LIMITS 201, GUARD_ACTION 192a, GUARD_GOLD_TIMER 181, PLAYER_COL 80c, PLAYER_ROW 80c, PTR2 78b, SHOULD_GUARD_MOVE_LEFT 215, SHOULD_GUARD_MOVE_RIGHT 217, and SHOULD_GUARD_MOVE_UP_OR_DOWN 219.

200 ⟨*defines* 4⟩+≡ (281) ◁197 220▷

```
GUARD_LEFT_COL_LIMIT      EQU     $5A
GUARD_RIGHT_COL_LIMIT     EQU     $5B
```

Defines:
GUARD_LEFT_COL_LIMIT, used in chunks 201 and 215.
GUARD_RIGHT_COL_LIMIT, used in chunks 201 and 217.

201 ⟨*determine guard left right limits* 201⟩≡ (278)

```
        ORG     $743E
    DETERMINE_GUARD_LEFT_RIGHT_LIMITS:
        SUBROUTINE

        STX     GUARD_LEFT_COL_LIMIT
        STX     GUARD_RIGHT_COL_LIMIT
        STY     ROWNUM

    .loop:
        LDA     GUARD_LEFT_COL_LIMIT
        BEQ     .loop2

        LDY     ROWNUM
        ⟨set active row pointer PTR1 for Y 78c⟩
        LDY     GUARD_LEFT_COL_LIMIT
        DEY
        LDA     (PTR1),Y

        CMP     #SPRITE_BRICK
        BEQ     .loop2
        CMP     #SPRITE_STONE
        BEQ     .loop2

        CMP     #SPRITE_LADDER
        BEQ     .next
        CMP     #SPRITE_ROPE
        BEQ     .next

        LDY     ROWNUM
        CPY     #MAX_GAME_ROW
        BEQ     .next

        ⟨set background row pointer PTR2 for Y+1 80a⟩
        LDY     GUARD_LEFT_COL_LIMIT
        DEY
        LDA     (PTR2),Y

        CMP     #SPRITE_BRICK
        BEQ     .next
        CMP     #SPRITE_STONE
        BEQ     .next
        CMP     #SPRITE_LADDER
        BNE     .end_loop

    .next:
        DEC     GUARD_LEFT_COL_LIMIT
        BPL     .loop

    .end_loop:
```

```
    DEC     GUARD_LEFT_COL_LIMIT

.loop2:
    LDA     GUARD_RIGHT_COL_LIMIT
    CMP     #MAX_GAME_COL
    BEQ     .end

    LDY     ROWNUM
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     GUARD_RIGHT_COL_LIMIT
    INY
    LDA     (PTR1),Y

    CMP     #SPRITE_BRICK
    BEQ     .end
    CMP     #SPRITE_STONE
    BEQ     .end

    CMP     #SPRITE_LADDER
    BEQ     .next_loop2
    CMP     #SPRITE_ROPE
    BEQ     .next_loop2

    LDY     ROWNUM
    CPY     #MAX_GAME_ROW
    BEQ     .next_loop2

    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     GUARD_RIGHT_COL_LIMIT
    INY
    LDA     (PTR2),Y

    CMP     #SPRITE_BRICK
    BEQ     .next_loop2
    CMP     #SPRITE_STONE
    BEQ     .next_loop2
    CMP     #SPRITE_LADDER
    BNE     .end_loop2

.next_loop2:
    INC     GUARD_RIGHT_COL_LIMIT
    BPL     .loop2

.end_loop2:
    INC     GUARD_RIGHT_COL_LIMIT

.end:
    RTS
```

Defines:
DETERMINE_GUARD_LEFT_RIGHT_LIMITS, used in chunk 198.

Uses GUARD_LEFT_COL_LIMIT 200, GUARD_RIGHT_COL_LIMIT 200, PTR1 78b, PTR2 78b, and ROWNUM 34a.

204 ⟨*try guard move left* 204⟩≡ (278)

```
      ORG     $6FBC
  TRY_GUARD_MOVE_LEFT:
      SUBROUTINE

      LDY     GUARD_LOC_ROW
      ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
      LDX     GUARD_X_ADJ
      CPX     #$03
      BCS     .move_left           ; horizontal adjustment > 0

      ; horizontal adjustment <= 0
      LDY     GUARD_LOC_COL
      BEQ     .store_guard_data       ; Can't go any more left

      DEY
      LDA     (PTR1),Y
      CMP     #SPRITE_GUARD
      BEQ     .store_guard_data
      CMP     #SPRITE_STONE
      BEQ     .store_guard_data
      CMP     #SPRITE_BRICK
      BEQ     .store_guard_data

      LDA     (PTR2),Y
      CMP     #SPRITE_TRAP
      BNE     .move_left

  .store_guard_data:
      JMP     STORE_GUARD_DATA        ; tailcall

  .move_left:
      JSR     GET_GUARD_SPRITE_AND_COORDS
      JSR     ERASE_SPRITE_AT_PIXEL_COORDS
      JSR     NUDGE_GUARD_TOWARDS_EXACT_ROW
      LDA     #$FF
      STA     GUARD_FACING_DIRECTION      ; face left
      DEC     GUARD_X_ADJ
      BPL     .check_for_gold_pickup

      ; horizontal adjustment underflow
      JSR     GUARD_DROP_GOLD
      LDY     GUARD_LOC_COL
      LDA     (PTR2),Y
      CMP     #SPRITE_BRICK
      BNE     .store_sprite
      LDA     #SPRITE_EMPTY

  .store_sprite:
      STA     (PTR1),Y
```

```
    DEC     GUARD_LOC_COL
    DEY
    LDA     (PTR1),Y
    CMP     #SPRITE_PLAYER
    BNE     .place_guard_sprite

    ; kill player
    LSR     ALIVE

.place_guard_sprite:
    LDA     #SPRITE_GUARD
    STA     (PTR1),Y

    LDA     #$04
    STA     GUARD_X_ADJ     ; horizontal adjustment = +2
    BNE     .determine_anim_set           ; unconditional

.check_for_gold_pickup:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD

.determine_anim_set:
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_ROPE
    BEQ     .rope

    LDA     #$00
    LDX     #$02
    BNE     .inc_anim_state

.rope:
    LDA     #$03
    LDX     #$05

.inc_anim_state:
    JSR     INC_GUARD_ANIM_STATE
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA                ; tailcall
```

Defines:
TRY_GUARD_MOVE_LEFT, used in chunk 192b.
Uses ALIVE 111a, CHECK_FOR_GOLD_PICKED_UP_BY_GUARD 184, DRAW_SPRITE_AT_PIXEL_COORDS 41, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_GUARD_SPRITE_AND_COORDS 187b, GUARD_FACING_DIRECTION 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_X_ADJ 181, INC_GUARD_ANIM_STATE 185, NUDGE_GUARD_TOWARDS_EXACT_ROW 183, PTR1 78b, PTR2 78b, and STORE_GUARD_DATA 186.

206 ⟨*try guard move right* 206⟩≡ (278)

```
      ORG     $7047
  TRY_GUARD_MOVE_RIGHT:
      SUBROUTINE

      LDY     GUARD_LOC_ROW
      ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
      LDX     GUARD_X_ADJ
      CPX     #$02
      BCC     .move_right           ; horizontal adjustment < 0

      ; horizontal adjustment >= 0
      LDY     GUARD_LOC_COL
      CPY     #MAX_GAME_COL
      BEQ     .store_guard_data       ; Can't go any more right

      INY
      LDA     (PTR1),Y
      CMP     #SPRITE_GUARD
      BEQ     .store_guard_data
      CMP     #SPRITE_STONE
      BEQ     .store_guard_data
      CMP     #SPRITE_BRICK
      BEQ     .store_guard_data

      LDA     (PTR2),Y
      CMP     #SPRITE_TRAP
      BNE     .move_right

  .store_guard_data:
      JMP     STORE_GUARD_DATA        ; tailcall

  .move_right:
      JSR     GET_GUARD_SPRITE_AND_COORDS
      JSR     ERASE_SPRITE_AT_PIXEL_COORDS
      JSR     NUDGE_GUARD_TOWARDS_EXACT_ROW
      LDA     #$01
      STA     GUARD_FACING_DIRECTION      ; face right
      INC     GUARD_X_ADJ
      LDA     GUARD_X_ADJ
      CMP     #$05
      BCC     .check_for_gold_pickup

      ; horizontal adjustment overflow
      JSR     GUARD_DROP_GOLD
      LDY     GUARD_LOC_COL
      LDA     (PTR2),Y
      CMP     #SPRITE_BRICK
      BNE     .store_sprite
      LDA     #SPRITE_EMPTY
```

```
.store_sprite:
    STA     (PTR1),Y

    INC     GUARD_LOC_COL
    INY
    LDA     (PTR1),Y
    CMP     #SPRITE_PLAYER
    BNE     .place_guard_sprite

    ; kill player
    LSR     ALIVE

.place_guard_sprite:
    LDA     #SPRITE_GUARD
    STA     (PTR1),Y

    LDA     #$00
    STA     GUARD_X_ADJ     ; horizontal adjustment = -2
    BEQ     .determine_anim_set           ; unconditional

.check_for_gold_pickup:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD

.determine_anim_set:
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y
    CMP     #SPRITE_ROPE
    BEQ     .rope

    LDA     #$07
    LDX     #$09
    BNE     .inc_anim_state

.rope:
    LDA     #$0A
    LDX     #$0C

.inc_anim_state:
    JSR     INC_GUARD_ANIM_STATE
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA                ; tailcall
```

Defines:
TRY_GUARD_MOVE_RIGHT, used in chunk 192b.
Uses ALIVE 111a, CHECK_FOR_GOLD_PICKED_UP_BY_GUARD 184, DRAW_SPRITE_AT_PIXEL_COORDS 41, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_GUARD_SPRITE_AND_COORDS 187b, GUARD_FACING_DIRECTION 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_X_ADJ 181, INC_GUARD_ANIM_STATE 185, NUDGE_GUARD_TOWARDS_EXACT_ROW 183, PTR1 78b, PTR2 78b, and STORE_GUARD_DATA 186.

208 ⟨*try guard move up* 208⟩≡ (278)

```
    ORG     $6EA1
GUARD_DO_NOTHING:
    SUBROUTINE

    ; if GUARD_GOLD_TIMER > 0, GUARD_GOLD_TIMER++
    LDA     GUARD_GOLD_TIMER
    BEQ     .store_guard_data
    BMI     .store_guard_data
    INC     GUARD_GOLD_TIMER
.store_guard_data:
    JMP     STORE_GUARD_DATA

    ORG     $6EAC
TRY_GUARD_MOVE_UP:
    SUBROUTINE

    LDY     GUARD_Y_ADJ
    CPY     #$03
    BCS     .move_up        ; vertical adjustment > 0

    LDY     GUARD_LOC_ROW
    BEQ     GUARD_DO_NOTHING
    DEY
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR1),Y

    CMP     #SPRITE_BRICK
    BEQ     GUARD_DO_NOTHING
    CMP     #SPRITE_STONE
    BEQ     GUARD_DO_NOTHING
    CMP     #SPRITE_TRAP
    BEQ     GUARD_DO_NOTHING
    CMP     #SPRITE_GUARD
    BEQ     GUARD_DO_NOTHING

.move_up:
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    JSR     NUDGE_GUARD_TOWARDS_EXACT_COLUMN
    LDY     GUARD_LOC_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
    DEC     GUARD_Y_ADJ
    BPL     TRY_GUARD_MOVE_UP_check_for_gold

    ; vertical adjustment underflow
    JSR     GUARD_DROP_GOLD
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y
```

```
    CMP     #SPRITE_BRICK
    BNE     .set_active_sprite
    LDA     #SPRITE_EMPTY
.set_active_sprite:
    STA     (PTR1),Y

    DEC     GUARD_LOC_ROW
    LDY     GUARD_LOC_ROW
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR1),Y

    CMP     #SPRITE_PLAYER
    BNE     .set_guard_sprite

    ; kill player
    LSR     ALIVE

.set_guard_sprite:
    LDA     #SPRITE_GUARD
    STA     (PTR1),Y

    LDA     #$04
    STA     GUARD_Y_ADJ         ; vertical adjust = +2
    BNE     TRY_GUARD_MOVE_UP_inc_anim_state     ; unconditional

TRY_GUARD_MOVE_UP_check_for_gold:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_GUARD

TRY_GUARD_MOVE_UP_inc_anim_state:
    LDA     #$0E
    LDX     #$0F
    JSR     INC_GUARD_ANIM_STATE
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA
```

Defines:
GUARD_DO_NOTHING, never used.
TRY_GUARD_MOVE_UP, used in chunk 192b.
Uses ALIVE 111a, CHECK_FOR_GOLD_PICKED_UP_BY_GUARD 184, DRAW_SPRITE_AT_PIXEL_COORDS 41, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_GUARD_SPRITE_AND_COORDS 187b, GUARD_GOLD_TIMER 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_Y_ADJ 181, INC_GUARD_ANIM_STATE 185, NUDGE_GUARD_TOWARDS_EXACT_COLUMN 183, PTR1 78b, PTR2 78b, and STORE_GUARD_DATA 186.

210 ⟨*try guard move down* 210⟩≡ (278)

```
      ORG     $6F39
  TRY_GUARD_MOVE_DOWN:
      SUBROUTINE

      LDY     GUARD_Y_ADJ
      CPY     #$02
      BCC     .move_down        ; vertical adjustment < 0

      LDY     GUARD_LOC_ROW
      CPY     #MAX_GAME_ROW
      BCS     .store_guard_data
      INY
      ⟨set active row pointer PTR1 for Y 78c⟩
      LDY     GUARD_LOC_COL
      LDA     (PTR1),Y

      CMP     #SPRITE_STONE
      BEQ     .store_guard_data
      CMP     #SPRITE_GUARD
      BEQ     .store_guard_data
      CMP     #SPRITE_BRICK
      BNE     .move_down

  .store_guard_data:
      JMP     STORE_GUARD_DATA

  .move_down:
      JSR     GET_GUARD_SPRITE_AND_COORDS
      JSR     ERASE_SPRITE_AT_PIXEL_COORDS
      JSR     NUDGE_GUARD_TOWARDS_EXACT_COLUMN
      LDY     GUARD_LOC_ROW
      ⟨set active and background row pointers PTR1 and PTR2 for Y 79a⟩
      INC     GUARD_Y_ADJ
      LDA     GUARD_Y_ADJ
      CMP     #$05
      BCC     .check_for_gold

      ; vertical adjustment overflow
      JSR     GUARD_DROP_GOLD
      LDY     GUARD_LOC_COL
      LDA     (PTR2),Y

      CMP     #SPRITE_BRICK
      BNE     .set_active_sprite
      LDA     #SPRITE_EMPTY
  .set_active_sprite:
      STA     (PTR1),Y

      INC     GUARD_LOC_ROW
```

```
    LDY     GUARD_LOC_ROW
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR1),Y

    CMP     #SPRITE_PLAYER
    BNE     .set_guard_sprite

    ; kill player
    LSR     ALIVE

.set_guard_sprite:
    LDA     #SPRITE_GUARD
    STA     (PTR1),Y

    LDA     #$00
    STA     GUARD_Y_ADJ         ; vertical adjust = -2
    JMP     TRY_GUARD_MOVE_UP_inc_anim_state

.check_for_gold:
    JMP     TRY_GUARD_MOVE_UP_check_for_gold
```

Defines:
TRY_GUARD_MOVE_DOWN, used in chunk 192b.
Uses ALIVE 111a, ERASE_SPRITE_AT_PIXEL_COORDS 38, GET_GUARD_SPRITE_AND_COORDS 187b, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, GUARD_Y_ADJ 181, NUDGE_GUARD_TOWARDS_EXACT_COLUMN 183, PTR1 78b, PTR2 78b, and STORE_GUARD_DATA 186.

This routine is called whenever we move a guard and the horizontal or vertical adjustment under- or overflows. If and only if GUARD_GOLD_TIMER is zero, decrement GUARD_GOLD_TIMER, and if there is nothing at the guard location, then drop gold at the location.

212 ⟨*guard drop gold* 212⟩≡ (278)

```
        ORG     $753E
    GUARD_DROP_GOLD:
        SUBROUTINE

        LDA     GUARD_GOLD_TIMER
        BPL     .end
        INC     GUARD_GOLD_TIMER
        BNE     .end

        ; GUARD_GOLD_TIMER == 0
        LDY     GUARD_LOC_ROW
        STY     GAME_ROWNUM
        ⟨set background row pointer PTR2 for Y 78d⟩
        LDY     GUARD_LOC_COL
        STY     GAME_COLNUM
        LDA     (PTR2),Y

        CMP     #SPRITE_EMPTY
        BNE     .decrement_flag_0

        ; Put gold at location
        LDA     #SPRITE_GOLD
        STA     (PTR2),Y
        JSR     DRAW_SPRITE_PAGE2
        LDY     GAME_ROWNUM
        LDX     GAME_COLNUM
        JSR     GET_SCREEN_COORDS_FOR
        LDA     #SPRITE_GOLD
        JMP     DRAW_SPRITE_AT_PIXEL_COORDS

    .decrement_flag_0:
        DEC     GUARD_GOLD_TIMER

    .end:
        RTS
```

Uses DRAW_SPRITE_AT_PIXEL_COORDS 41, DRAW_SPRITE_PAGE2 35, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_SCREEN_COORDS_FOR 31a, GUARD_GOLD_TIMER 181, GUARD_LOC_COL 181, GUARD_LOC_ROW 181, and PTR2 78b.

The PSUEDO_DISTANCE returns a distance measure between the player and the given A, X coordinate based on whether the point is above, below, or on the same row as the player row.

If the point is on the same row as the player, then the return value is the horizontal distance between the current guard and the point. Otherwise, if the point is above the player row, return 200 plus the vertical distance between the point and the player. Otherwise, the point is below the player row, so return 100 plus the vertical distance between the point and the player.

213 ⟨*pseudo distance* 213⟩≡ (278)

```
    ORG     $72D4
PSEUDO_DISTANCE:
    SUBROUTINE

    STA     TMP
    CMP     PLAYER_ROW
    BNE     .tmp_not_player_row

    ; TMP == PLAYER_ROW
    ; return | X - GUARD_LOC_COL |

    CPX     GUARD_LOC_COL
    BCC     .x_lt_guard_col

    ; X >= GUARD_LOC_COL
    TXA
    ; A = X - GUARD_LOC_COL
    SEC
    SBC     GUARD_LOC_COL
    RTS

.x_lt_guard_col:
    STX     TMP

    ; A = GUARD_LOC_COL - X
    LDA     GUARD_LOC_COL
    SEC
    SBC     TMP
    RTS

.tmp_not_player_row:
    ; If TMP >= PLAYER_ROW, return 200 + | TMP - PLAYER_ROW |
    ; otherwise return 100 + | TMP - PLAYER_ROW |

    BCC     .tmp_lt_player_row

    ; TMP >= PLAYER_ROW
    ; A = TMP - PLAYER_ROW + 200
    SEC
    SBC     PLAYER_ROW
```

```
    CLC
    ADC     #200
    RTS

.tmp_lt_player_row
    ; A = PLAYER_ROW - TMP + 100
    LDA     PLAYER_ROW
    SEC
    SBC     TMP
    CLC
    ADC     #100
    RTS
```

Defines:
PSUEDO_DISTANCE, never used.
Uses GUARD_LOC_COL 181, PLAYER_ROW 80c, and TMP 4.

215 ⟨*should guard move left* 215⟩≡ (278)

```
    ORG     $71A1
    SUBROUTINE
.return:
    RTS

SHOULD_GUARD_MOVE_LEFT:
    LDY     GUARD_LEFT_COL_LIMIT
    CPY     TMP_GUARD_COL
    BEQ     .return

    LDY     TMP_GUARD_ROW
    CPY     #MAX_GAME_ROW
    BEQ     .check_here

    ; Check below:

    ; Get background sprite at TMP_GUARD_ROW + 1, col = GUARD_LEFT_COL_LIMIT
    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     GUARD_LEFT_COL_LIMIT
    LDA     (PTR2),Y

    CMP     #SPRITE_BRICK
    BEQ     .check_here
    CMP     #SPRITE_STONE
    BEQ     .check_here

    LDX     GUARD_LEFT_COL_LIMIT
    LDY     TMP_GUARD_ROW
    JSR     GUARD_FIND_CANDIDATE_ROW_BELOW

    LDX     GUARD_LEFT_COL_LIMIT
    JSR     PSEUDO_DISTANCE
    CMP     BEST_GUARD_DIST
    BCS     .check_here        ; dist >= BEST_GUARD_DIST?

    ; dist < BEST_GUARD_DIST
    STA     BEST_GUARD_DIST     ; dist
    LDA     #GUARD_ACTION_MOVE_LEFT
    STA     GUARD_ACTION

.check_here:
    LDY     TMP_GUARD_ROW
    BEQ     .next

    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     GUARD_LEFT_COL_LIMIT
    LDA     (PTR2),Y

    CMP     #SPRITE_LADDER
```

```
    BNE     .next

    ; Ladder here
    LDY     TMP_GUARD_ROW
    LDX     GUARD_LEFT_COL_LIMIT
    JSR     GUARD_FIND_CANDIDATE_ROW_ABOVE

    LDX     GUARD_LEFT_COL_LIMIT
    JSR     PSEUDO_DISTANCE
    CMP     BEST_GUARD_DIST
    BCS     .next        ; dist >= BEST_GUARD_DIST?

    ; dist < BEST_GUARD_DIST
    STA     BEST_GUARD_DIST     ; dist
    LDA     #GUARD_ACTION_MOVE_LEFT
    STA     GUARD_ACTION

.next:
    INC     GUARD_LEFT_COL_LIMIT
    JMP     SHOULD_GUARD_MOVE_LEFT
```

Defines:
SHOULD_GUARD_MOVE_LEFT, used in chunk 198.
Uses BEST_GUARD_DIST 197, GUARD_ACTION 192a, GUARD_FIND_CANDIDATE_ROW_ABOVE 224, GUARD_FIND_CANDIDATE_ROW_BELOW 221, GUARD_LEFT_COL_LIMIT 200, and PTR2 78b.

217 ⟨*should guard move right* 217⟩≡ (278)

```
    ORG     $720B
    SUBROUTINE
.return:
    RTS

SHOULD_GUARD_MOVE_RIGHT:
    LDY     GUARD_RIGHT_COL_LIMIT
    CPY     TMP_GUARD_COL
    BEQ     .return

    LDY     TMP_GUARD_ROW
    CPY     #MAX_GAME_ROW
    BEQ     .check_here

    ; Check below:

    ; Get background sprite at TMP_GUARD_ROW + 1, col = GUARD_RIGHT_COL_LIMIT
    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     GUARD_RIGHT_COL_LIMIT
    LDA     (PTR2),Y

    CMP     #SPRITE_BRICK
    BEQ     .check_here
    CMP     #SPRITE_STONE
    BEQ     .check_here

    LDX     GUARD_RIGHT_COL_LIMIT
    LDY     TMP_GUARD_ROW
    JSR     GUARD_FIND_CANDIDATE_ROW_BELOW

    LDX     GUARD_RIGHT_COL_LIMIT
    JSR     PSEUDO_DISTANCE
    CMP     BEST_GUARD_DIST
    BCS     .check_here        ; dist >= BEST_GUARD_DIST?

    ; dist < BEST_GUARD_DIST
    STA     BEST_GUARD_DIST     ; dist
    LDA     #GUARD_ACTION_MOVE_RIGHT
    STA     GUARD_ACTION

.check_here:
    LDY     TMP_GUARD_ROW
    BEQ     .next

    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     GUARD_RIGHT_COL_LIMIT
    LDA     (PTR2),Y

    CMP     #SPRITE_LADDER
```

```
    BNE     .next

    ; Ladder here
    LDY     TMP_GUARD_ROW
    LDX     GUARD_RIGHT_COL_LIMIT
    JSR     GUARD_FIND_CANDIDATE_ROW_ABOVE

    LDX     GUARD_RIGHT_COL_LIMIT
    JSR     PSEUDO_DISTANCE
    CMP     BEST_GUARD_DIST
    BCS     .next        ; dist >= BEST_GUARD_DIST?

    ; dist < BEST_GUARD_DIST
    STA     BEST_GUARD_DIST     ; dist
    LDA     #GUARD_ACTION_MOVE_RIGHT
    STA     GUARD_ACTION

.next:
    DEC     GUARD_RIGHT_COL_LIMIT
    JMP     SHOULD_GUARD_MOVE_RIGHT
```

Defines:
SHOULD_GUARD_MOVE_RIGHT, used in chunk 198.
Uses BEST_GUARD_DIST 197, GUARD_ACTION 192a, GUARD_FIND_CANDIDATE_ROW_ABOVE 224, GUARD_FIND_CANDIDATE_ROW_BELOW 221, GUARD_RIGHT_COL_LIMIT 200, and PTR2 78b.

219 ⟨*should guard move up or down* 219⟩≡ (278)

```
        ORG    $7275
    SHOULD_GUARD_MOVE_UP_OR_DOWN:
        SUBROUTINE

        LDY     TMP_GUARD_ROW
        CPY     #MAX_GAME_ROW
        BEQ     .should_guard_move_up

        ⟨set background row pointer PTR2 for Y+1 80a⟩
        LDY     TMP_GUARD_COL
        LDA     (PTR2),Y

        CMP     #SPRITE_BRICK
        BEQ     .should_guard_move_up
        CMP     #SPRITE_STONE
        BEQ     .should_guard_move_up

        LDX     TMP_GUARD_COL
        LDY     TMP_GUARD_ROW
        JSR     GUARD_FIND_CANDIDATE_ROW_BELOW
        LDX     TMP_GUARD_COL
        JSR     PSEUDO_DISTANCE
        CMP     BEST_GUARD_DIST
        BCS     .should_guard_move_up
        STA     BEST_GUARD_DIST
        LDA     #GUARD_ACTION_MOVE_DOWN
        STA     GUARD_ACTION

    .should_guard_move_up:
        LDY     TMP_GUARD_ROW
        BEQ     .end

        ⟨set background row pointer PTR2 for Y 78d⟩
        LDY     TMP_GUARD_COL
        LDA     (PTR2),Y

        CMP     #SPRITE_LADDER
        BNE     .end
        ; ladder

        LDX     TMP_GUARD_COL
        LDY     TMP_GUARD_ROW
        JSR     GUARD_FIND_CANDIDATE_ROW_ABOVE
        LDX     TMP_GUARD_COL
        JSR     PSEUDO_DISTANCE
        CMP     BEST_GUARD_DIST
        BCS     .end
        STA     BEST_GUARD_DIST
        LDA     #GUARD_ACTION_MOVE_UP
```

```
    STA     GUARD_ACTION

.end:
    RTS
```

Defines:
SHOULD_GUARD_MOVE_UP_OR_DOWN, used in chunk 198.
Uses BEST_GUARD_DIST 197, GUARD_ACTION 192a, GUARD_FIND_CANDIDATE_ROW_ABOVE 224, GUARD_FIND_CANDIDATE_ROW_BELOW 221, and PTR2 78b.

220 ⟨*defines* 4⟩+≡ (281) ◁200 227▷

```
CHECK_CURR_TMP_ROW  EQU     $5C
CHECK_TMP_COL       EQU     $5D
CHECK_TMP_ROW       EQU     $5E
```

Defines:
CHECK_TMP_COL, used in chunks 221 and 224.
CHECK_TMP_ROW, used in chunks 221 and 224.

The GUARD_FIND_CANDIDATE_ROW_BELOW is called when determining if the guard should move down. It scans below the guard for a candidate row. Upon entry, we store X and Y in CHECK_TMP_COL and CHECK_TMP_ROW. Next, we scan from CHECK_TMP_ROW to the bottommost game row.

For each row:

- If the background sprite below the test coordinate is brick or stone, return CHECK_TMP_ROW.
- Otherwise, if the sprite below the test coordinate is not empty:
  - If we're not all the way to the left:
    - If there's a rope to the left, or if there's a brick, stone, or ladder below left then if this is below or on the same row as the PLAYER_ROW, return CHECK_TMP_ROW.
  - If we're not all the way to the right:
    - If there's a rope to the right, or if there's a brick, stone, or ladder below right then if this is below or on the same row as the PLAYER_ROW, return CHECK_TMP_ROW.

And if we haven't returned in the loop, just return the MAX_GAME_ROW.

221 ⟨*guard find candidate row below* 221⟩≡ (278)

```
    ORG     $739A
    SUBROUTINE
.return_tmp_row:
    LDA     CHECK_TMP_ROW
    RTS

GUARD_FIND_CANDIDATE_ROW_BELOW:
    STY     CHECK_TMP_ROW
    STX     CHECK_TMP_COL

    ; for CHECK_TMP_ROW = Y; CHECK_TMP_ROW <= MAX_GAME_ROW; CHECK_TMP_ROW++

.loop:
    ; if background sprite below tmp coords is brick or stone, return tmp row.

    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     CHECK_TMP_COL
    LDA     (PTR2),Y

    CMP     #SPRITE_BRICK
    BEQ     .return_tmp_row
    CMP     #SPRITE_STONE
    BEQ     .return_tmp_row

    ; Not brick or stone below
    ; if background sprite at tmp coords is empty, then next tmp row.
```

```
    LDY     CHECK_TMP_ROW
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     CHECK_TMP_COL
    LDA     (PTR2),Y

    CMP     #SPRITE_EMPTY
    BEQ     .next

    CPY     #$00
    BEQ     .check_right      ; cannot check to left

    ; if background sprite to left of tmp coords is rope,
    ; then tmp_row -> curr_tmp_row
    DEY
    LDA     (PTR2),Y        ; Check to left

    CMP     #SPRITE_ROPE
    BEQ     .store_as_curr_tmp_row

    ; if background sprite to left and below tmp coords is brick, stone, or ladder,
    ; then tmp_row -> curr_tmp_row
    LDY     CHECK_TMP_ROW
    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     CHECK_TMP_COL
    DEY
    LDA     (PTR2),Y

    CMP     #SPRITE_BRICK
    BEQ     .store_as_curr_tmp_row
    CMP     #SPRITE_STONE
    BEQ     .store_as_curr_tmp_row
    CMP     #SPRITE_LADDER
    BNE     .check_right

    ; Otherwise check right

.store_as_curr_tmp_row:
    ; Store tmp row as curr tmp row, and if at or below player, return curr tmp row.
    LDY     CHECK_TMP_ROW
    STY     CHECK_CURR_TMP_ROW
    CPY     PLAYER_ROW
    BCS     .return_curr_tmp_row
    ; CHECK_TMP_ROW < PLAYER_ROW

.check_right:
    LDY     CHECK_TMP_COL
    CPY     #MAX_GAME_COL
    BCS     .next           ; can't check right
```

```
    ; if background sprite to right is rope,
    ; then tmp_row -> curr_tmp_row
    INY
    LDA     (PTR2),Y

    CMP     #SPRITE_ROPE
    BEQ     .store_as_curr_tmp_row_2

    ; if background sprite to right and below tmp coords is brick, stone, or ladder,
    ; then tmp_row -> curr_tmp_row
    LDY     CHECK_TMP_ROW
    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     CHECK_TMP_COL
    INY
    LDA     (PTR2),Y

    CMP     #SPRITE_BRICK
    BEQ     .store_as_curr_tmp_row_2
    CMP     #SPRITE_LADDER
    BEQ     .store_as_curr_tmp_row_2
    CMP     #SPRITE_STONE
    BNE     .next
    ; Brick, ladder, or stone.

.store_as_curr_tmp_row_2:
    LDY     CHECK_TMP_ROW
    STY     CHECK_CURR_TMP_ROW
    CPY     PLAYER_ROW
    BCS     .return_curr_tmp_row
    ; CHECK_TMP_ROW < PLAYER_ROW

.next:
    INC     CHECK_TMP_ROW
    LDY     CHECK_TMP_ROW
    CPY     #MAX_GAME_ROW+1
    BCS     .return_max_game_row
    JMP     .loop

.return_max_game_row:
    LDA     #MAX_GAME_ROW
    RTS

.return_curr_tmp_row:
    LDA     CHECK_CURR_TMP_ROW
    RTS
```

Defines:
GUARD_FIND_CANDIDATE_ROW_BELOW, used in chunks 215, 217, and 219.
Uses CHECK_TMP_COL 220, CHECK_TMP_ROW 220, PLAYER_ROW 80c, and PTR2 78b.

224 ⟨*guard find candidate row above* 224⟩≡ (278)

```
    ORG     $72FD
    SUBROUTINE
.not_ladder:
    LDA     CHECK_TMP_ROW         ; row
    RTS

GUARD_FIND_CANDIDATE_ROW_ABOVE:
    STY     CHECK_TMP_ROW         ; row
    STX     CHECK_TMP_COL         ; col

.loop:
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     CHECK_TMP_COL         ; col
    LDA     (PTR2),Y    ; sprite on background

    CMP     #SPRITE_LADDER
    BNE     .not_ladder     ; no ladder at row, col -> just return row.

    ; There is a ladder at row, col
    DEC     CHECK_TMP_ROW     ; row--     ; up one
    LDY     CHECK_TMP_COL     ; col
    BEQ     .at_leftmost

    DEY             ; to left (col-1)
    LDA     (PTR2),Y

    ; To left of ladder is brick, stone, or ladder: .blocked_on_left
    CMP     #SPRITE_BRICK
    BEQ     .blocked_on_left
    CMP     #SPRITE_STONE
    BEQ     .blocked_on_left
    CMP     #SPRITE_LADDER
    BEQ     .blocked_on_left

    LDY     CHECK_TMP_ROW     ; row (that is now up one)
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     CHECK_TMP_COL     ; col
    DEY
    LDA     (PTR2),Y    ; sprite on background

    CMP     #SPRITE_ROPE
    BNE     .at_leftmost

    ; There is a rope above the ladder

.blocked_on_left:
    ; If row <= PLAYER_ROW (on or above player row), return row
    LDY     CHECK_TMP_ROW     ; row
    STY     SCRATCH_5C
```

```
    CPY     PLAYER_ROW
    BCC     .return_scratch_5C
    BEQ     .return_scratch_5C

.at_leftmost:
    LDY     CHECK_TMP_COL     ; col
    CPY     #MAX_GAME_COL
    BEQ     .at_rightmost

    ; Look at background sprite below and to the right
    LDY     CHECK_TMP_ROW     ; row
    ⟨set background row pointer PTR2 for Y+1 80a⟩
    LDY     CHECK_TMP_COL
    INY
    LDA     (PTR2),Y        ; get background sprite at row+1, col+1

    ; Below and to the right of ladder is brick, stone, or ladder: .blocked_below
    CMP     #SPRITE_BRICK
    BEQ     .blocked_below
    CMP     #SPRITE_STONE
    BEQ     .blocked_below
    CMP     #SPRITE_LADDER
    BEQ     .blocked_below

    ; Look at background sprite to the right
    LDY     CHECK_TMP_ROW     ; row
    ⟨set background row pointer PTR2 for Y 78d⟩
    LDY     CHECK_TMP_COL     ; col
    INY
    LDA     (PTR2),Y    ; get background sprite at row, col+1

    CMP     #SPRITE_ROPE
    BNE     .at_rightmost

    ; There is a rope to the right of the ladder

.blocked_below:
    ; If row <= PLAYER_ROW (on or above player row), return row
    LDY     CHECK_TMP_ROW     ; row
    STY     SCRATCH_5C
    CPY     PLAYER_ROW
    BCC     .return_scratch_5C
    BEQ     .return_scratch_5C

.at_rightmost:
    ; If row < 1, return row, otherwise loop.
    LDY     CHECK_TMP_ROW     ; row
    CPY     #$01
    BCC     .return_Y
    JMP     .loop
```

```
.return_Y:
    TYA
    RTS

.return_scratch_5C:
    LDA     SCRATCH_5C
    RTS
```

Defines:
GUARD_FIND_CANDIDATE_ROW_ABOVE, used in chunks 215, 217, and 219.
Uses CHECK_TMP_COL 220, CHECK_TMP_ROW 220, PLAYER_ROW 80c, PTR2 78b, and SCRATCH_5C 4.

Chapter 10

Disk routines

Lode Runner contains a copy of DOS 3.3's RWTS in order for it to access the disk. Because it does not access files, but rather just sectors of data, Lode Runner does not need the file manager part of DOS. So, the area `B600-BFFF` is taken by DOS routines.

The standard DOS I/O Control Block (IOB) and Device Characteristics Table (DCT) are used. Further details can be read in Beneath Apple DOS.

There is one patch that seems to have been applied. DOS has a routine called `FORMDSK` located at `BE0D` which is supposed to jump to `DSKFORM` at `BE46`, which formats a disk. Lode Runner's copy of DOS changes the jump target to `8E00`, Lode Runner's `FORMAT_PATCH` routine.

The reason for this is to prevent formatting the Load Runner master disk via the level editor.

The routine moves to track 0, and keeps reading bytes off the disk until either it finds a sequence of `D4 D5 D6`, or times out. If it finds the sequence, we've detected that this disk we're trying to format is the Lode Runner master disk, and we should error out and go back to the level editor.

Otherwise we're OK to proceed with disk formatting.

227 ⟨*defines* 4⟩+≡ (281) ◁220 229▷

```
SEEKABS     EQU     $B9A0
DSKFORM     EQU     $BEAF
```

228 ⟨*format patch* 228⟩≡ (278)

```
        ORG     $8E00
    FORMAT_PATCH:
        LDA     #$44
        STA     $0478       ; Used by DOS as current track.
        LDA     #$00
        JSR     SEEKABS     ; DOS routine to move disk to track in A.
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        LDA     #$20
        STA     $4F

    .await_D4_D5_D6:
        DEY
        BNE     .read_byte1
        DEC     $4F
        BNE     .read_byte1
        JMP     DSKFORM     ; DOS routine to format disk

        HEX     EA EA

    .read_byte1:
        LDA     $C08C,X
        BPL     .read_byte1

    .check_for_D4:
        CMP     #$D4
        BNE     .await_D4_D5_D6
        NOP

    .read_byte2:
        LDA     $C08C,X
        BPL     .read_byte2

    .check_for_D5:
        CMP     #$D5
        BNE     .check_for_D4
        NOP

    .read_byte3:
        LDA     $C08C,X
        BPL     .read_byte3

        CMP     #$D6
        BNE     .check_for_D5
```

```
    LDA     $C088,X         ; Turn motor off
    JSR     DONT_MANIPULATE_MASTER_DISK
    JMP     START_LEVEL_EDITOR
```

Defines:
  FORMAT_PATCH, never used.
Uses DONT_MANIPULATE_MASTER_DISK 237b and START_LEVEL_EDITOR 259.

229 ⟨*defines* 4⟩+≡ (281) ◁227 232▷

```
DOS_IOB                         EQU     $B7E8
IOB_SLOTNUMx16                  EQU     $B7E9
IOB_DRIVE_NUM                   EQU     $B7EA
IOB_VOLUME_NUMBER_EXPECTED      EQU     $B7EB
IOB_TRACK_NUMBER                EQU     $B7EC
IOB_SECTOR_NUMBER               EQU     $B7ED
IOB_DEVICE_CHARACTERISTICS_TABLE_PTR        EQU     $B7EE   ; 2 bytes
IOB_READ_WRITE_BUFFER_PTR       EQU     $B7F0   ; 2 bytes
IOB_UNUSED                      EQU     $B7F2
IOB_BYTE_COUNT_FOR_PARTIAL_SECTOR   EQU     $B7F3
IOB_COMMAND_CODE                EQU     $B7F4
IOB_RETURN_CODE                 EQU     $B7F5
IOB_LAST_ACCESS_VOLUME          EQU     $B7F6
IOB_LAST_ACCESS_SLOTx16         EQU     $B7F7
IOB_LAST_ACCESS_DRIVE           EQU     $B7F8

DCT_DEVICE_TYPE                 EQU     $B7FB
DCT_PHASES_PER_TRACK            EQU     $B7FC
DCT_MOTOR_ON_TIME_COUNT         EQU     $B7FD   ; 2 bytes
```

Defines:
  DCT_DEVICE_TYPE, never used.
  DCT_MOTOR_ON_TIME_COUNT, never used.
  DCT_PHASES_PER_TRACK, never used.
  DOS_IOB, used in chunks 108 and 231.
  IOB_BYTE_COUNT_FOR_PARTIAL_SECTOR, never used.
  IOB_COMMAND_CODE, used in chunks 108, 231, and 241.
  IOB_DEVICE_CHARACTERISTICS_TABLE_PTR, never used.
  IOB_DRIVE_NUM, never used.
  IOB_LAST_ACCESS_DRIVE, never used.
  IOB_LAST_ACCESS_SLOTx16, never used.
  IOB_LAST_ACCESS_VOLUME, never used.
  IOB_READ_WRITE_BUFFER_PTR, used in chunks 108, 231, and 241.
  IOB_RETURN_CODE, never used.
  IOB_SECTOR_NUMBER, used in chunks 108, 231, and 241.
  IOB_SLOTNUMx16, never used.
  IOB_TRACK_NUMBER, used in chunks 108, 231, and 241.
  IOB_UNUSED, never used.
  IOB_VOLUME_NUMBER_EXPECTED, used in chunks 108 and 231.

ACCESS_HI_SCORE_DATA_FROM_DISK reads or writes—depending on A, where 1 is read and 2 is write—the high score table from disk at track 12 sector 15 into HI_SCORE_TABLE. We then compare the 11 bytes of HI_SCORE_DATA_MARKER to where they are supposed to be in the table.

If the marker doesn't match, then we return 0, indicating that the disk doesn't have a high score table.

If the marker does match, but the very last byte in the table is nonzero, then we return 1, indicating that this is a master disk (so its level data shouldn't be touched), otherwise we return -1, this being a data disk.

230 ⟨*tables* 9⟩+≡ (281) ◁192b 240▷

```
    ORG     $63A8
HI_SCORE_DATA_MARKER:
    ; Spells out "LODE RUNNER".
    HEX     CC CF C4 C5 A0 D2 D5 CE CE C5 D2
```

Defines:
HI_SCORE_DATA_MARKER, used in chunks 231 and 241.

231 ⟨*access hi score data* 231⟩≡ (278)

```
        ORG     $6359
    ACCESS_HI_SCORE_DATA_FROM_DISK:
        SUBROUTINE

        STA     IOB_COMMAND_CODE
        LDA     #$0C
        STA     IOB_TRACK_NUMBER
        LDA     #$0F
        STA     IOB_SECTOR_NUMBER
        LDA     #<HI_SCORE_DATA
        STA     IOB_READ_WRITE_BUFFER_PTR
        LDA     #>HI_SCORE_DATA
        STA     IOB_READ_WRITE_BUFFER_PTR+1
        LDA     #$00
        STA     IOB_VOLUME_NUMBER_EXPECTED
        LDY     #<DOS_IOB
        LDA     #>DOS_IOB
        JSR     INDIRECT_RWTS
        BCC     .no_error
        JMP     RESET_GAME

    .no_error:
        LDY     #$0A
        LDA     #$00
        STA     MASK0       ; temp storage

    .loop:
        LDA     HI_SCORE_DATA+244,Y
        EOR     HI_SCORE_DATA_MARKER,Y
        ORA     MASK0
        STA     MASK0
        DEY
        BPL     .loop

        LDA     MASK0
        BEQ     .all_zero_data

        LDA     #$00
        RTS

    .all_zero_data:
        LDA     #$01
        LDX     $1FFF
        BNE     .end
        LDA     #$FF

    .end:
        RTS
```

Defines:

ACCESS_HI_SCORE_DATA_FROM_DISK, used in chunks 133b, 233, 238, 241, 244, 246, 268, and 277a.
Uses DOS_IOB 229, HI_SCORE_DATA 119a, HI_SCORE_DATA_MARKER 230, INDIRECT_RWTS 245b, IOB_COMMAND_CODE 229, IOB_READ_WRITE_BUFFER_PTR 229, IOB_SECTOR_NUMBER 229, IOB_TRACK_NUMBER 229, IOB_VOLUME_NUMBER_EXPECTED 229, and MASK0 34a.

RECORD_HI_SCORE_DATA_TO_DISK records the player's score to disk if the player's score belongs on the high score list. It also handles getting the player's initials.

232 ⟨*defines* 4⟩+≡ (281) ◁229 239▷

```
HIGH_SCORE_INITIALS_INDEX       EQU     $824D
HI_SCORE_TARGET_INDEX           EQU     $56     ; aliased with TMP_GUARD_ROW
```

Defines:
HIGH_SCORE_INITIALS_INDEX, used in chunk 233.

233 ⟨*record hi score data* 233⟩≡ (278)

```
        ORG     $84C8
    RECORD_HI_SCORE_DATA_TO_DISK:
        SUBROUTINE

        LDA     $9D
        BEQ     .end

        LDA     SCORE
        ORA     SCORE+1
        ORA     SCORE+2
        ORA     SCORE+3
        BEQ     .end

        LDA     #$01
        JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read table
        ; Return value of 0 means the hi score marker wasn't present,
        ; so don't write the hi score table.
        BEQ     .end

        LDY     #$01
    .loop:
        LDX     HI_SCORE_TABLE_OFFSETS,Y
        LDA     LEVELNUM
        CMP     HI_SCORE_DATA+3,X       ; level
        BCC     .next
        BNE     .record_it

        LDA     SCORE+3
        CMP     HI_SCORE_DATA+4,X
        BCC     .next
        BNE     .record_it

        LDA     SCORE+2
        CMP     HI_SCORE_DATA+5,X
        BCC     .next
        BNE     .record_it

        LDA     SCORE+1
        CMP     HI_SCORE_DATA+6,X
        BCC     .next
        BNE     .record_it

        LDA     SCORE
        CMP     HI_SCORE_DATA+7,X
        BCC     .next
        BNE     .record_it

    .next:
        INY
```

```
    CPY     #$0B
    BCC     .loop

.end:
    RTS

.record_it:
    CPY     #$0A
    BEQ     .write_here
    STY     HI_SCORE_TARGET_INDEX

    ; Move the table rows to make room at index HI_SCORE_TARGET_INDEX
    LDY     #$09
.loop2:
    LDX     HI_SCORE_TABLE_OFFSETS,Y

    ; Move 8 bytes of hi score data
    LDA     #$08
    STA     ROW_COUNT       ; temporary counter
.loop3:
    LDA     HI_SCORE_DATA,X
    STA     HI_SCORE_DATA+8,X
    INX
    DEC     ROW_COUNT
    BNE     .loop3

    CPY     HI_SCORE_TARGET_INDEX
    BEQ     .write_here
    DEY
    BNE     .loop2

.write_here:
    LDX     HI_SCORE_TABLE_OFFSETS,Y
    LDA     #$A0
    STA     HI_SCORE_DATA,X
    STA     HI_SCORE_DATA+1,X
    STA     HI_SCORE_DATA+2,X
    LDA     LEVELNUM
    STA     HI_SCORE_DATA+3,X
    LDA     SCORE+3
    STA     HI_SCORE_DATA+4,X
    LDA     SCORE+2
    STA     HI_SCORE_DATA+5,X
    LDA     SCORE+1
    STA     HI_SCORE_DATA+6,X
    LDA     SCORE
    STA     HI_SCORE_DATA+7,X
    STY     WIPE0               ; temporary
    LDA     HI_SCORE_TABLE_OFFSETS,Y
    STA     .rd_loc+1
```

```
    STA     .wr_loc+1
    JSR     HI_SCORE_SCREEN

    LDA     #$40
    STA     DRAW_PAGE
    LDA     WIPE0
    CLC
    ADC     #$04
    STA     GAME_ROWNUM
    LDA     #$07
    STA     GAME_COLNUM

    LDX     #$00
    STX     HIGH_SCORE_INITIALS_INDEX
.get_initial_from_player:
    LDX     HIGH_SCORE_INITIALS_INDEX
.rd_loc:
    LDA     HI_SCORE_DATA,X     ; fixed up to add offset from above
    JSR     CHAR_TO_SPRITE_NUM
    JSR     WAIT_FOR_KEY
    STA     KBDSTRB
    CMP     #$8D
    BEQ     .return_pressed
    CMP     #$88                ; backspace/back arrow
    BNE     .other_key_pressed

    ; backspace pressed
    LDX     KBD_ENTRY_INDEX
    BEQ     .beep       ; can't backspace/back arrow past the beginning

    DEC     HIGH_SCORE_INITIALS_INDEX
    DEC     GAME_COLNUM
    JMP     .get_initial_from_player

.other_key_pressed:
    CMP     #$95            ; fwd arrow
    BNE     .check_for_allowed_chars
    LDX     KBD_ENTRY_INDEX
    CPX     #$02
    BEQ     .beep       ; can't fwd arrow past the end

    INC     GAME_COLNUM
    INC     KBD_ENTRY_INDEX
    JMP     .get_initial_from_player

.check_for_allowed_chars
    CMP     #$AE        ; period allowed
    BEQ     .put_char
    CMP     #$A0        ; space allowed
    BEQ     .put_char
```

```
    CMP     #$C1
    BCC     .beep       ; can't be less than 'A'
    CMP     #$DB
    BCS     .beep       ; can't be greater than 'Z'

.put_char
    LDY     KBD_ENTRY_INDEX
.wr_loc:
    STA     HI_SCORE_DATA,Y     ; fixed up to add offset from above
    JSR     PUT_CHAR
    INC     KBD_ENTRY_INDEX
    LDA     KBD_ENTRY_INDEX
    CMP     #$03
    BCC     .get_initial_from_player

    DEC     KBD_ENTRY_INDEX
    DEC     GAME_COLNUM
    JMP     .get_initial_from_player

.beep:
    JSR     BEEP
    JMP     .get_initial_from_player

.return_pressed:
    LDA     #$20
    STA     DRAW_PAGE
    LDA     #$02
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; write hi score table
    JMP     LONG_DELAY

    ORG     $824D
KBD_ENTRY_INDEX:
    HEX     60
```

Defines:
KBD_ENTRY_INDEX, used in chunk 74.
RECORD_HI_SCORE_DATA_TO_DISK, used in chunk 250.
Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231, BEEP 56, CHAR_TO_SPRITE_NUM 44, DRAW_PAGE 45, GAME_COLNUM 34a, GAME_ROWNUM 34a, HI_SCORE_DATA 119a, HI_SCORE_SCREEN 119b, HI_SCORE_TABLE_OFFSETS 121a, HIGH_SCORE_INITIALS_INDEX 232, KBDSTRB 68b, LEVELNUM 52, PUT_CHAR 46a, ROW_COUNT 25c, SCORE 50b, WAIT_FOR_KEY 70, and WIPE0 91.

237a ⟨*bad data disk* 237a⟩≡ (278)

```
      ORG     $8106
  BAD_DATA_DISK:
      SUBROUTINE

      JSR     CLEAR_HGR2
      LDA     #$40
      STA     DRAW_PAGE
      LDA     #$00
      STA     GAME_COLNUM
      STA     GAME_ROWNUM

      ; "DISKETTE IN DRIVE IS NOT A\r"
      ; "LODE RUNNER DATA DISK."
      JSR     PUT_STRING
      HEX     C4 C9 D3 CB C5 D4 D4 C5 A0 C9 CE A0 C4 D2 C9 D6
      HEX     C5 A0 C9 D3 A0 CE CF D4 A0 C1 8D CC CF C4 C5 A0
      HEX     D2 D5 CE CE C5 D2 A0 C4 C1 D4 C1 A0 C4 C9 D3 CB
      HEX     AE 00

      JMP     HIT_KEY_TO_CONTINUE
```

Defines:
BAD_DATA_DISK, used in chunks 238, 244, 246, and 268.
Uses CLEAR_HGR2 5, DRAW_PAGE 45, GAME_COLNUM 34a, GAME_ROWNUM 34a, HIT_KEY_TO_CONTINUE 73a, and PUT_STRING 47.

237b ⟨*dont manipulate master disk* 237b⟩≡ (278)

```
      ORG     $8098
  DONT_MANIPULATE_MASTER_DISK:
      SUBROUTINE

      JSR     CLEAR_HGR2
      LDA     #$40
      STA     DRAW_PAGE
      LDA     #$00
      STA     GAME_COLNUM
      STA     GAME_ROWNUM

      ; "USER NOT ALLOWED TO\r"
      ; "MANIPULATE MASTER DISKETTE."
      JSR     PUT_STRING
      HEX     D5 D3 C5 D2 A0 CE CF D4 A0 C1 CC CC CF D7 C5 C4
      HEX     A0 D4 CF 8D CD C1 CE C9 D0 D5 CC C1 D4 C5 A0 CD
      HEX     C1 D3 D4 C5 D2 A0 C4 C9 D3 CB C5 D4 D4 C5 AE 00

      ; fallthrough to HIT_KEY_TO_CONTINUE
```

Defines:
DONT_MANIPULATE_MASTER_DISK, used in chunks 228, 238, and 268.
Uses CLEAR_HGR2 5, DRAW_PAGE 45, GAME_COLNUM 34a, GAME_ROWNUM 34a, HIT_KEY_TO_CONTINUE 73a, and PUT_STRING 47.

The level editor has a routine to check for a valid data disk, meaning it has a high score table and is not the master disk. In case of a disk that is not a valid data disk, we abort the current editor operation, dumping the user right into the level editor by jumping to START_LEVEL_EDITOR. Otherwise we jump to RETURN_FROM_SUBROUTINE, which apparently saved a byte over having a local RTS instruction.

238 ⟨*check for valid data disk* 238⟩≡ (278)

```
    ORG     $807F
CHECK_FOR_VALID_DATA_DISK:
    SUBROUTINE

    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read table
    CMP     #$00        ; bad table
    BNE     .check_for_master_disk

    JSR     BAD_DATA_DISK
    JMP     START_LEVEL_EDITOR

.check_for_master_disk:
    CMP     #$01        ; master disk
    BNE     RETURN_FROM_SUBROUTINE

    JSR     DONT_MANIPULATE_MASTER_DISK
    JMP     START_LEVEL_EDITOR
```

Defines:
CHECK_FOR_VALID_DATA_DISK, used in chunks 261a, 262, and 265.
Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231, BAD_DATA_DISK 237a, DONT_MANIPULATE_MASTER_DISK 237b, RETURN_FROM_SUBROUTINE 73a, and START_LEVEL_EDITOR 259.

Initializing a disk first DOS formats it. This zeros out all data on all tracks and sectors. Once that's done, we write track 0 sector 0 with the data from DISK_BOOT_SECTOR_DATA. Then we read the Volume Table of Contents (VTOC) at track 17 sector 0, which will contain all zeros because of the initial format. We then stick SAVED_VTOC_DATA in the disk buffer and write it to the VTOC. We do the same thing with the catalog sector at track 17 sector 15 and SAVED_FILE_DESCRIPTIVE_ENTRY_DATA.

The final step is to create a blank sector at track 12 sector 15, with the special "LODE RUNNER" marker HI_SCORE_DATA_MARKER near the end.

239 ⟨*defines* 4⟩+≡ (281) ◁232 245c▷

```
    ORG     $1DB2
DISK_BOOT_SECTOR_DATA:
    HEX     01 20 58 FC 20 93 FE 20 89 FE A0 00 B9 34 08 F0
    HEX     0E 20 F0 FD C9 8D D0 04 A9 09 85 24 C8 D0 ED A6
    HEX     2B 9D 88 C0 8A 4A 4A 4A 4A 09 C0 8D 33 08 20 0C
    HEX     FD 4C 00 C6 8D 8D 8D 8D 8D 8D 8D CC CF C4 C5 A0
    HEX     D2 D5 CE CE C5 D2 A0 C4 C1 D4 C1 A0 C4 C9 D3 CB
    HEX     BA 8D AD AD AD AD AD AD AD AD AD AD AD AD AD AD
    HEX     AD AD AD AD AD AD AD AD 8D 8D C4 C9 D3 CB C5 D4
    HEX     D4 C5 A0 D7 C9 CC CC A0 CE CF D4 A0 C2 CF CF D4
    HEX     8D 8D A0 C9 CE D3 C5 D2 D4 A0 CE C5 D7 A0 C4 C9
    HEX     D3 CB A0 C1 CE C4 8D A0 C8 C9 D4 A0 C1 A0 CB C5
    HEX     D9 A0 D4 CF A0 D2 C5 C2 CF CF D4 8D 8D A0 A0 A0
    HEX     A0 A0 A0 A0 A0 A0 A0 00 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Defines:
DISK_BOOT_SECTOR_DATA, used in chunk 241.

240 ⟨*tables* 9⟩+≡ (281) ◁230 247▷

```
      ORG     $8250
  SAVED_VTOC_DATA:
      HEX     60 02 11 0F 04 00 00 FE 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      HEX     00 00 00 00 00 00 00 00 7A 00 00 00 00 00 00 00
      HEX     00 FF FF 00 00 23 0F 00 01

      ORG     $8289
  SAVED_FILE_DESCRIPTIVE_ENTRY_DATA:
      HEX     22 ; Track of first track/sector list sector (T34)
      HEX     0F ; Sector of first track/sector list sector (S15)
      HEX     88 ; File type and flags: locked, S-type file
      ; File name: "^H^H^H^H^H^HLODE RUNNER DATA DISK  "
      HEX     88 88 88 88 88 88 88 CC CF C4 C5 A0 D2 D5 CE CE
      HEX     C5 D2 A0 C4 C1 D4 C1 A0 C4 C9 D3 CB A0 A0
```

Defines:
SAVED_FILE_DESCRIPTIVE_ENTRY_DATA, used in chunk 241.
SAVED_VTOC_DATA, used in chunks 241 and 262.

241 ⟨*editor initialize disk* 241⟩≡ (278)

```
      ORG     $7D5D
  EDITOR_INITIALIZE_DISK:
      SUBROUTINE

      ; "\r"
      ; ">>INITIALIZE\r"
      ; "  THIS FORMATS THE DISKETTE\r"
      ; "  FOR USER CREATED LEVELS.\r"
      ; "  (CAUTION. IT ERASES THE\r"
      ; "   ENTIRE DISKETTE FIRST)\r"
      ; "\r"
      ; "  ARE YOU SURE (Y/N) "
      JSR     PUT_STRING
      HEX     8D BE BE C9 CE C9 D4 C9 C1 CC C9 DA C5 8D A0 A0
      HEX     D4 C8 C9 D3 A0 C6 CF D2 CD C1 D4 D3 A0 D4 C8 C5
      HEX     A0 C4 C9 D3 CB C5 D4 D4 C5 8D A0 A0 C6 CF D2 A0
      HEX     D5 D3 C5 D2 A0 C3 D2 C5 C1 D4 C5 C4 A0 CC C5 D6
      HEX     C5 CC D3 AE 8D A0 A0 A8 C3 C1 D5 D4 C9 CF CE AE
      HEX     A0 C9 D4 A0 C5 D2 C1 D3 C5 D3 A0 D4 C8 C5 8D A0
      HEX     A0 A0 C5 CE D4 C9 D2 C5 A0 C4 C9 D3 CB C5 D4 D4
      HEX     C5 A0 C6 C9 D2 D3 D4 A9 8D 8D A0 A0 C1 D2 C5 A0
      HEX     D9 CF D5 A0 D3 D5 D2 C5 A0 A8 D9 AF CE A9 A0 00

      JSR     EDITOR_WAIT_FOR_KEY
      CMP     #$D9        ; Y
      BNE     .end

      NOP     ; NOP x 15
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP

      LDA     DISK_LEVEL_LOC
      PHA

      ; Format the disk
      LDA     #DISK_ACCESS_FORMAT
      JSR     ACCESS_COMPRESSED_LEVEL_DATA
```

```
    ; Write the boot sector (T0S0)
    LDA     #<DISK_BOOT_SECTOR_DATA
    STA     IOB_READ_WRITE_BUFFER_PTR
    LDA     #>DISK_BOOT_SECTOR_DATA
    STA     IOB_READ_WRITE_BUFFER_PTR+1
    LDA     #$00
    STA     IOB_SECTOR_NUMBER
    STA     IOB_TRACK_NUMBER
    LDA     #$02
    STA     IOB_COMMAND_CODE
    JSR     ACCESS_DISK_OR_RESET_GAME   ; write T0S0 with DISK_BOOT_SECTOR_DATA.

    ; Read the VTOC (T17S0)
    LDA     #$E0
    STA     DISK_LEVEL_LOC              ; ends up being T17S0 (the VTOC)
    LDA     #DISK_ACCESS_READ
    JSR     ACCESS_COMPRESSED_LEVEL_DATA

    ; Copy from SAVED_VTOC_DATA to DISK_BUFFER and write it.
    LDY     #$37
.loop:
    LDA     SAVED_VTOC_DATA+1,Y
    STA     DISK_BUFFER,Y
    DEY
    BPL     .loop

    LDA     #$02
    JSR     ACCESS_COMPRESSED_LEVEL_DATA

    ; Read the first catalog sector (T17S15)
    LDA     #$EF
    STA     DISK_LEVEL_LOC
    LDA     #DISK_ACCESS_READ
    JSR     ACCESS_COMPRESSED_LEVEL_DATA

    ; Copy from SAVED_FILE_DESCRIPTIVE_ENTRY_DATA the first file descriptive
    ; entry to DISK_BUFFER and write it.
    LDY     #$20
.loop2:
    LDA     SAVED_FILE_DESCRIPTIVE_ENTRY_DATA,Y
    STA     DISK_BUFFER+11,Y
    DEY
    BPL     .loop2

    ; Write it back
    LDA     #DISK_ACCESS_WRITE
    JSR     ACCESS_COMPRESSED_LEVEL_DATA

    ; Read the high score sector
```

```
    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK

    ; Copy from HI_SCORE_DATA_MARKER and write it.
    LDY     #$0A
.loop3:
    LDA     HI_SCORE_DATA_MARKER,Y
    STA     $1FF4,Y
    DEY
    BPL     .loop3

    ; Write it back
    LDA     #$02
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK

    PLA
    STA     DISK_LEVEL_LOC
.end:
    JMP     EDITOR_COMMAND_LOOP
```

Defines:
EDITOR_INITIALIZE_DISK, used in chunk 258.
Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231, DISK_BOOT_SECTOR_DATA 239, EDITOR_COMMAND_LOOP 259, EDITOR_WAIT_FOR_KEY 72, HI_SCORE_DATA_MARKER 230, IOB_COMMAND_CODE 229, IOB_READ_WRITE_BUFFER_PTR 229, IOB_SECTOR_NUMBER 229, IOB_TRACK_NUMBER 229, PUT_STRING 47, SAVED_FILE_DESCRIPTIVE_ENTRY_DATA 240, and SAVED_VTOC_DATA 240.

To clear the high score table from a disk, we first read the sector where the high score table is supposed to be, and check to see if the buffer is a good high score table. If so, we zero out the first 80 bytes (the 10 high score entries) and write that back to disk.

If the disk didn't contain a good high score table, we display the BAD_DATA_DISK message and abort.

244 ⟨*editor clear high scores* 244⟩≡ (278)

```
    ORG     $7E75
EDITOR_CLEAR_HIGH_SCORES:
    SUBROUTINE

    ; "\r"
    ; ">>CLEAR SCORE FILE\r"
    ; "  THIS CLEARS THE HIGH\r"
    ; "  SCORE FILE OF ALL\r"
    ; "  ENTRIES.\r"
    ; "\r"
    ; "  ARE YOU SURE (Y/N) "
    JSR     PUT_STRING
    HEX     8D BE BE C3 CC C5 C1 D2 A0 D3 C3 CF D2 C5 A0 C6
    HEX     C9 CC C5 8D A0 A0 D4 C8 C9 D3 A0 C3 CC C5 C1 D2
    HEX     D3 A0 D4 C8 C5 A0 C8 C9 C7 C8 8D A0 A0 D3 C3 CF
    HEX     D2 C5 A0 C6 C9 CC C5 A0 CF C6 A0 C1 CC CC 8D A0
    HEX     A0 C5 CE D4 D2 C9 C5 D3 AE 8D 8D A0 A0 C1 D2 C5
    HEX     A0 D9 CF D5 A0 D3 D5 D2 C5 A0 A8 D9 AF CE A9 A0
    HEX     00

    JSR     EDITOR_WAIT_FOR_KEY
    CMP     #$D9        ; 'Y'
    BNE     .end

    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read table
    CMP     #$00
    BNE     .good_disk
    JSR     BAD_DATA_DISK
    JMP     START_LEVEL_EDITOR

.good_disk:
    LDY     #$4F
    LDA     #$00

.loop:
    STA     HI_SCORE_DATA,Y
    DEY
    BPL     .loop

    LDA     #$02
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; write table
```

```
.end:
    JMP     EDITOR_COMMAND_LOOP
```

Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231, BAD_DATA_DISK 237a, EDITOR_COMMAND_LOOP 259, EDITOR_WAIT_FOR_KEY 72, HI_SCORE_DATA 119a, PUT_STRING 47, SCORE 50b, and START_LEVEL_EDITOR 259.

10.1 Initialization

245a ⟨*rwts targets* 245a⟩≡ (278)

```
INDIRECT_TARGET             EQU     $36     ; Init with DEFAULT_INDIRECT_TARGET
DISABLE_INTS_CALL_RWTS_PTR  EQU     $38     ; Init with DISABLE_INTS_CALL_RWTS
DISABLE_INTS_CALL_RWTS      EQU     $B7B5
```

Defines:
DISABLE_INTS_CALL_RWTS, used in chunk 245b.
DISABLE_INTS_CALL_RWTS_PTR, used in chunk 246.
INDIRECT_TARGET, used in chunks 245b, 246, and 259.

245b ⟨*indirect call* 245b⟩≡ (278)

```
    ORG     $63A5
INDIRECT_RWTS:
    SUBROUTINE
    JMP     (INDIRECT_TARGET)

    ORG     $8E50
DEFAULT_INDIRECT_TARGET:
    SUBROUTINE
    JMP     DISABLE_INTS_CALL_RWTS
```

Defines:
INDIRECT_RWTS, used in chunk 231.
Uses DISABLE_INTS_CALL_RWTS 245a and INDIRECT_TARGET 245a.

245c ⟨*defines* 4⟩+≡ (281) ◁239 261b▷

```
GUARD_PATTERN_OFFSET        EQU     $97
```

Defines:
GUARD_PATTERN_OFFSET, used in chunks 115b, 141a, 246, and 250.

246 ⟨*Initialize game data* 246⟩≡ (278)

```
        ORG     $6056

    INIT_GAME_DATA:
        LDA     #0
        STA     SCORE
        STA     SCORE+1
        STA     SCORE+2
        STA     SCORE+3
        STA     GUARD_PATTERN_OFFSET
        STA     WIPE_MODE       ; WIPE_MODE = SCORE = $97 = 0
        STA     $53
        STA     $AB
        STA     $A8             ; $53 = $AB = $A8 = 0
        LDA     #$9B            ; 155
        STA     $A9             ; $A9 = 155
        LDA     #5
        STA     LIVES           ; LIVES = 5
        LDA     GAME_MODE
        LSR
        ; if GAME_MODE was 0 or 1 (i.e. not displaying high score screen or splash screen),
        ; play the game.
        BEQ     .put_status_and_start_game

        ; We were displaying the high score screen or splash screen
        LDA     #1
        JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; Read hi score data
        CMP     #$00
        BNE     .set_rwts_target
        JSR     BAD_DATA_DISK
        JMP     RESET_GAME

    .set_rwts_target:
        LDA     $1FFF
        BNE     .use_dos_target
        LDA     INDIRECT_TARGET
        LDX     INDIRECT_TARGET+1
        BNE     .store_rwts_addr

    .use_dos_target:
        LDA     DISABLE_INTS_CALL_RWTS_PTR
        LDX     DISABLE_INTS_CALL_RWTS_PTR+1

    .store_rwts_addr:
        STA     RWTS_ADDR
        STX     RWTS_ADDR+1

    .put_status_and_start_game:
        JSR     PUT_STATUS
        STA     TXTPAGE1
```

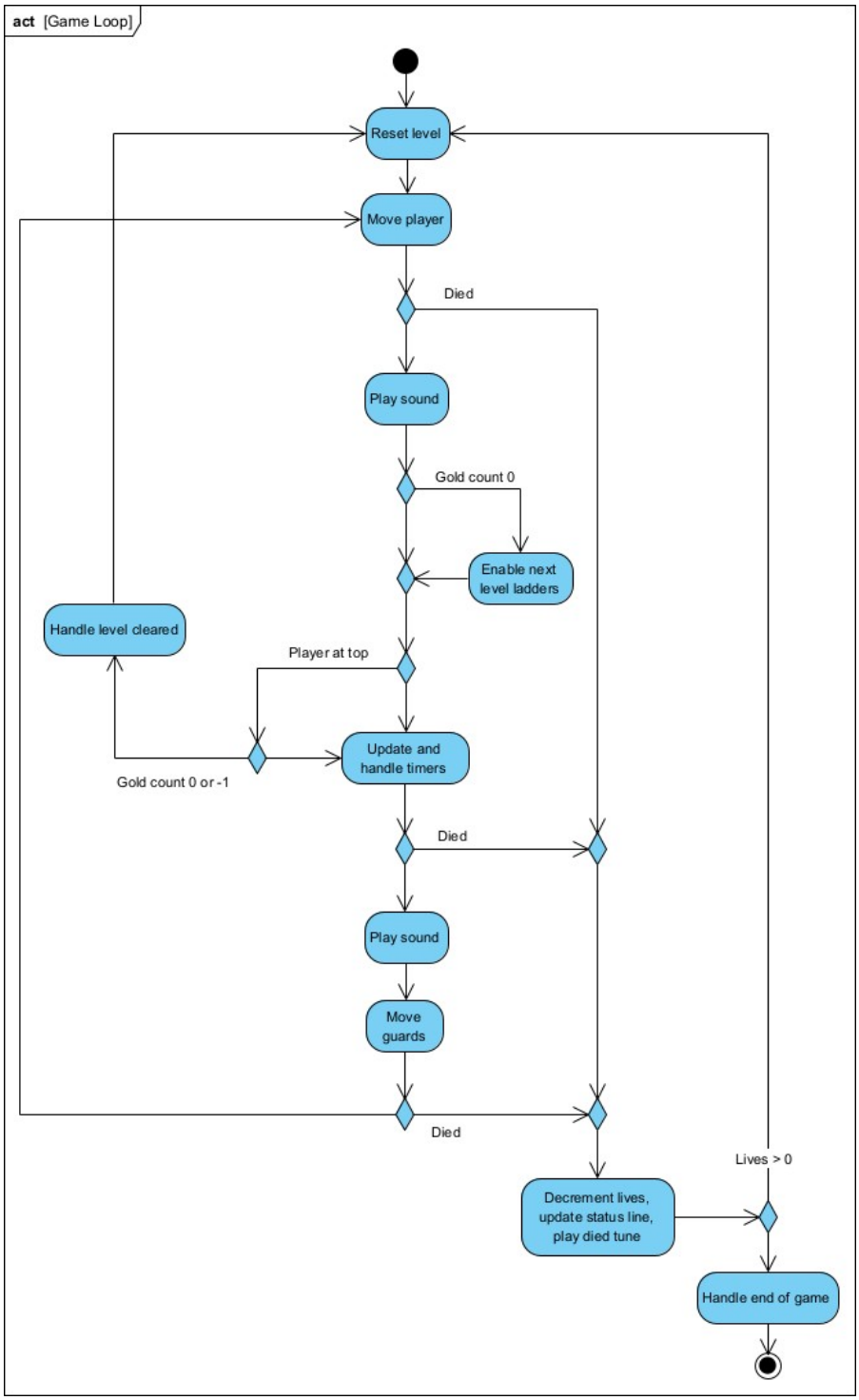
Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231, BAD_DATA_DISK 237a, DISABLE_INTS_CALL_RWTS_PTR 245a, GAME_MODE 106a, GUARD_PATTERN_OFFSET 245c, INDIRECT_TARGET 245a, LIVES 52, PUT_STATUS 53, SCORE 50b, TXTPAGE1 130a, and WIPE_MODE 88.

247 ⟨*tables* 9⟩+≡ (281) ◁240 252▷

```
        ORG     $6CA7
GUARD_PATTERNS_LIST:
        HEX     00 00 00
        HEX     00 01 01
        HEX     01 01 01
        HEX     01 03 01
        HEX     01 03 03
        HEX     03 03 03
        HEX     03 03 07
        HEX     03 07 07
        HEX     07 07 07
        HEX     07 07 0F
        HEX     07 0F 0F
        HEX     0F 0F 0F
```

Defines:
GUARD_PATTERNS_LIST, used in chunk 250.

The game loop, which runs in both attract mode and in play mode, effectively implements the following flowchart:


act [Game Loop]
Reset level
Move player
Died
Play sound
Gold count 0
Enable next level ladders
Handle level cleared
Player at top
Update and handle timers
Gold count 0 or -1
Died
Play sound
Move guards
Died
Lives > 0
Decrement lives, update status line, play died tune
Handle end of game

250 ⟨*game loop* 250⟩≡ (278)

```
    ORG     $609F

.start_game:
    LDX     #$01
    JSR     LOAD_LEVEL
    LDA     #$00
    STA     KEY_COMMAND
    STA     KEY_COMMAND_LR
    LDA     GAME_MODE
    LSR
    ; if GAME_MODE was 0 or 1 (i.e. not displaying high score screen
    ; or splash screen), play the game.
    BEQ     .play_game

    ; When GAME_MODE is 2:
    JSR     WAIT_KEY
    LDA     PLAYER_COL
    STA     GAME_COLNUM
    LDA     PLAYER_ROW
    STA     GAME_ROWNUM
    LDA     #SPRITE_PLAYER
    JSR     WAIT_FOR_KEY_WITH_CURSOR_PAGE_1

.play_game:
    LDX     #$00
    STX     DIG_DIRECTION
    STX     NOTE_INDEX

    LDA     GUARD_PATTERN_OFFSET
    CLC
    ADC     GUARD_COUNT         ; GUARD_COUNT + $97 can't be greater than 8.
    TAY
    LDX     TIMES_3_TABLE,Y     ; X = 3 * Y (goes up to Y=8)
    LDA     GUARD_PATTERNS_LIST,X
    STA     GUARD_PATTERNS
    LDA     GUARD_PATTERNS_LIST+1,X
    STA     GUARD_PATTERNS+1
    LDA     GUARD_PATTERNS_LIST+2,X
    STA     GUARD_PATTERNS+2

    LDY     GUARD_PATTERN_OFFSET
    LDA     $621D,Y
    STA     GUARD_GOLD_TIMER_START_VALUE

.game_loop:
    JSR     MOVE_PLAYER
    LDA     ALIVE
    BEQ     .died
```

```
    JSR     PLAY_SOUND

    LDA     GOLD_COUNT
    BNE     .check_player_reached_top
    JSR     ENABLE_NEXT_LEVEL_LADDERS

.check_player_reached_top:
    LDA     PLAYER_ROW
    BNE     .not_at_top
    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BNE     .not_at_top

    ; Reached top of screen
    LDA     GOLD_COUNT
    BEQ     .level_cleared
    CMP     #$FF
    BEQ     .level_cleared      ; level cleared if GOLD_COUNT == 0 or -1.

.not_at_top:
    JSR     HANDLE_TIMERS
    LDA     ALIVE
    BEQ     .died
    JSR     PLAY_SOUND
    JSR     MOVE_GUARDS
    LDA     ALIVE
    BEQ     .died
    BNE     .game_loop

.level_cleared:
    INC     LEVELNUM
    INC     DISK_LEVEL_LOC
    INC     LIVES
    BNE     .lives_incremented
    DEC     LIVES               ; LIVES doesn't overflow.

.lives_incremented:
    ; Increment score by 1500, playing an ascending tune while doing so.
    LDX     #$0F
    STX     SCRATCH_5C

.loop2:
    LDY     #$01
    LDA     #$00
    JSR     ADD_AND_UPDATE_SCORE    ; SCORE += 100
    JSR     APPEND_LEVEL_CLEARED_NOTE
    JSR     APPEND_LEVEL_CLEARED_NOTE
    JSR     APPEND_LEVEL_CLEARED_NOTE
    DEC     SCRATCH_5C
    BNE     .loop2
```

```
.start_game_:
    JMP     .start_game

.died:
    DEC     LIVES
    JSR     PUT_STATUS_LIVES
    JSR     LOAD_SOUND_DATA
    HEX     02 40 02 40 03 50 03 50 04 60 04 60 05 70 05 70
    HEX     06 80 06 80 07 90 07 90 08 A0 08 A0 09 B0 09 B0
    HEX     0A C0 0A C0 0B D0 0B D0 0C E0 0C E0 0D F0 0D F0
    HEX     00

.play_died_tune:
    JSR     PLAY_SOUND
    BCS     .play_died_tune

    LDA     GAME_MODE
    LSR
    BEQ     .restore_enable_sound     ; If GAME_MODE is 0 or 1

    LDA     LIVES
    BNE     .start_game_    ; We can still play.

    ; Game over
    JSR     RECORD_HI_SCORE_DATA_TO_DISK
    JSR     SPINNING_GAME_OVER
    BCS     .key_pressed
```

Uses ADD_AND_UPDATE_SCORE 51, ALIVE 111a, APPEND_LEVEL_CLEARED_NOTE 63, DIG_DIRECTION 166, ENABLE_NEXT_LEVEL_LADDERS 179, GAME_COLNUM 34a, GAME_MODE 106a, GAME_ROWNUM 34a, GOLD_COUNT 81d, GUARD_COUNT 81d, GUARD_GOLD_TIMER_START_VALUE 181, GUARD_PATTERN_OFFSET 245c, GUARD_PATTERNS 190, GUARD_PATTERNS_LIST 247, HANDLE_TIMERS 126, KEY_COMMAND 138a, KEY_COMMAND_LR 138a, LEVELNUM 52, LIVES 52, LOAD_LEVEL 111b, LOAD_SOUND_DATA 58, MOVE_GUARDS 191a, MOVE_PLAYER 175, NOTE_INDEX 57, PLAY_SOUND 62, PLAYER_COL 80c, PLAYER_ROW 80c, PLAYER_Y_ADJ 84b, PUT_STATUS_LIVES 53, RECORD_HI_SCORE_DATA_TO_DISK 233, SCORE 50b, SCRATCH_5C 4, TIMES_3_TABLE 252, WAIT_FOR_KEY_WITH_CURSOR_PAGE_1 71, and WAIT_KEY 69a.

252 ⟨*tables* 9⟩+≡ (281) ◁247 253▷

```
    ORG     $6214
TIMES_3_TABLE:
    HEX     00 03 06 09 0C 0F 12 15 18
```

Defines:
TIMES_3_TABLE, used in chunk 250.

253 ⟨*tables* 9⟩+≡ (281) ◁252 258▷

```
    ORG     $8C35
TABLE0:
    HEX     80 80 80 80 80 80 80 80 80 80 80 80 80 80
TABLE1:
    HEX     C0 AA D5 AA D5 AA D5 AA D5 AA D5 AA D5 80
TABLE2:
    HEX     90 80 80 80 80 80 80 80 80 80 80 80 80 82
TABLE3:
    HEX     90 AA D1 A2 D5 A8 85 A8 C5 A2 D4 A2 95 82
TABLE4:
    HEX     90 82 91 A2 C5 A8 80 88 C5 A2 94 A0 90 82
TABLE5:
    HEX     90 82 90 A2 C4 A8 80 88 C5 A2 94 A0 90 82
TABLE6:
    HEX     90 82 90 A2 C4 A8 81 88 C4 A2 D4 A0 95 82
TABLE7:
    HEX     90 A2 D1 A2 C4 88 80 88 C4 A2 84 A0 85 82
TABLE8:
    HEX     90 82 91 A2 C4 88 80 88 C4 AA 84 A0 85 82
TABLE9:
    HEX     90 82 91 A2 C4 88 80 88 C4 8A 84 A0 91 82
TABLE10:
    HEX     90 AA 91 A2 C4 A8 85 A8 85 82 D4 A2 91 82

    ORG     $8CCF
ADDRESS_TABLE:
    WORD    TABLE0-13
    WORD    TABLE1-13
    WORD    TABLE2-13
    WORD    TABLE3-13
    WORD    TABLE4-13
    WORD    TABLE5-13
    WORD    TABLE6-13
    WORD    TABLE7-13
    WORD    TABLE8-13
    WORD    TABLE9-13
    WORD    TABLE10-13
```

Defines:
ADDRESS_TABLE, used in chunk 256.

254 ⟨*anims* 254⟩≡ (278)

```
        ORG     $8B1A
    SPINNING_GAME_OVER:
        SUBROUTINE

        LDA     #$01
        STA     ANIM_COUNT
        LDA     #$20
        STA     HGR_PAGE

    .loop:
        JSR     ANIM5
        JSR     ANIM4
        JSR     ANIM3
        JSR     ANIM2
        JSR     ANIM1
        JSR     ANIM0
        JSR     ANIM1
        JSR     ANIM2
        JSR     ANIM3
        JSR     ANIM4
        JSR     ANIM5
        JSR     ANIM10
        JSR     ANIM9
        JSR     ANIM8
        JSR     ANIM7
        JSR     ANIM6
        JSR     ANIM7
        JSR     ANIM8
        JSR     ANIM9
        JSR     ANIM10
        LDA     ANIM_COUNT
        CMP     #100
        BCC     .loop

        JSR     ANIM5
        JSR     ANIM4
        JSR     ANIM3
        JSR     ANIM2
        JSR     ANIM1
        JSR     ANIM0
        CLC
        RTS

        ORG     $8B7A
    ANIM0:
        JSR     SHOW_ANIM_LINE
        HEX     00 01 02 03 04 05 06 07 08 09 0A 02 01 00
    ANIM1:
        JSR     SHOW_ANIM_LINE
```

```
    HEX     00 00 01 02 03 04 05 07 09 0A 02 01 00 00
ANIM2:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 01 02 03 04 09 0A 02 01 00 00 00
ANIM3:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 01 02 03 0A 02 01 00 00 00 00
ANIM4:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 00 01 03 0A 01 00 00 00 00 00
ANIM5:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 00 00 01 01 00 00 00 00 00 00
ANIM6:
    JSR     SHOW_ANIM_LINE
    HEX     00 01 02 0A 09 08 07 06 05 04 03 02 01 00
ANIM7:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 01 02 0A 09 07 05 04 03 02 01 00 00
ANIM8:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 01 02 0A 09 04 03 02 01 00 00 00
ANIM9:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 01 02 0A 03 02 01 00 00 00 00
ANIM10:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 00 01 0A 03 01 00 00 00 00 00
```

Uses ANIM_COUNT 256, HGR_PAGE 28b, and SHOW_ANIM_LINE 256.

256 ⟨*show anim line* 256⟩≡ (278)

```
        ORG     $8CE5
    SHOW_ANIM_LINE:
        SUBROUTINE

        PLA
        STA     TMP_PTR
        PLA
        STA     TMP_PTR+1           ; store "return" addr

        ; Fill 14 rows of pixel data from row 0x51 (81) through 0x5E (94).
        LDY     #$50
        STY     GAME_ROWNUM
        BNE     .next       ; unconditional

    .loop:
        JSR     ROW_TO_ADDR
        LDY     #$00
        LDA     (TMP_PTR),Y
        ASL
        TAX
        LDA     ADDRESS_TABLE,X
        STA     .loop2+1
        LDA     ADDRESS_TABLE+1,X   ; groups of 14 bytes
        STA     .loop2+2
        LDY     #$0D

        ; Copy 13 bytes of pixel data onto screen from
        ; addr+14 to addr+26
    .loop2:
        LDA     $8D08,Y             ; fixed up from above
        STA     (ROW_ADDR),Y        ; pixel data
        INY
        CPY     #$1B
        BCC     .loop2              ; Y < 27

        ; Next row
    .next:
        JSR     INCREMENT_TMP_PTR
        INC     GAME_ROWNUM
        LDY     GAME_ROWNUM
        CPY     #$5F
        BCC     .loop

        LDX     ANIM_COUNT
        LDY     #$FF
    .delay:
        DEY
        BNE     .delay
        DEX
```

```
    BNE     .delay
    INC     ANIM_COUNT

    LDA     INPUT_MODE
    CMP     #KEYBOARD_MODE
    BEQ     .check_for_keypress
    LDA     BUTN1
    BMI     .input_detected
    LDA     BUTN0
    BMI     .input_detected

.check_for_keypress:
    LDA     KBD
    BMI     .input_detected
    RTS

    ; Skip the rest of the big animation.
.input_detected:
    PLA
    PLA
    SEC
    LDA     KBD
    STA     KBDSTRB
    RTS

    ORG     $8D4B
ANIM_COUNT:
    HEX     9D

    ORG     $8D4C
INCREMENT_TMP_PTR:
    SUBROUTINE

    INC     TMP_PTR
    BNE     .end
    INC     TMP_PTR+1
.end:
    RTS
```

Defines:
ANIM_COUNT, used in chunk 254.
INCREMENT_TMP_PTR, never used.
SHOW_ANIM_LINE, used in chunk 254.
Uses ADDRESS_TABLE 253, BUTN0 66, BUTN1 66, GAME_ROWNUM 34a, INPUT_MODE 66, KBD 68b, KBDSTRB 68b, ROW_ADDR 28b, ROW_TO_ADDR 28c, and TMP_PTR 4.

Chapter 11

Level editor

258 ⟨*tables* 9⟩+≡ (281) ◁253 267▷

```
      ORG     $7C4D
  EDITOR_KEYS:
      ; P (Play level)
      ; C (Clear level)
      ; E (Edit level)
      ; M (Move level)
      ; I (Initialize disk)
      ; S (clear high Scores)
      HEX     D0 C3 C5 CD C9 D3 00    ; P C E M I S
  EDITOR_ROUTINE_ADDRESS:
      WORD    EDITOR_PLAY_LEVEL-1
      WORD    EDITOR_CLEAR_LEVEL-1
      WORD    EDITOR_EDIT_LEVEL-1
      WORD    EDITOR_MOVE_LEVEL-1
      WORD    EDITOR_INITIALIZE_DISK-1
      WORD    EDITOR_CLEAR_HIGH_SCORES-1
```

Defines:
  EDITOR_KEYS, used in chunk 259.
  EDITOR_ROUTINE_ADDRESS, used in chunk 259.
Uses EDITOR_CLEAR_LEVEL 261a, EDITOR_EDIT_LEVEL 264, EDITOR_INITIALIZE_DISK 241, EDITOR_MOVE_LEVEL 262, and EDITOR_PLAY_LEVEL 263.

259 ⟨*level editor* 259⟩≡ (278)

```
        ORG     $7B84
    LEVEL_EDITOR:
        SUBROUTINE

        LDA     #$00
        STA     SCORE
        STA     SCORE+1
        STA     SCORE+2
        STA     SCORE+3

        LDA     INDIRECT_TARGET
        STA     RWTS_ADDR
        LDA     INDIRECT_TARGET+1
        STA     RWTS_ADDR+1

        LDA     #$05
        STA     LIVES
        STA     GAME_MODE
        LDA     INPUT_MODE
        STA     READ_SAVED_INPUT_MODE+1

        LDA     #KEYBOARD_MODE
        STA     INPUT_MODE

        STA     TXTPAGE1

        LDA     DISK_LEVEL_LOC
        CMP     #$96
        BCC     START_LEVEL_EDITOR
        LDA     #$00
        STA     DISK_LEVEL_LOC

    START_LEVEL_EDITOR:
        JSR     CLEAR_HGR1
        LDA     #$20
        STA     DRAW_PAGE
        LDA     #$00
        STA     GAME_COLNUM
        STA     GAME_ROWNUM

        ; "  LODE RUNNER BOARD EDITOR\r
        ; "---------------------------\r
        ; "  <ESC> ABORTS ANY COMMAND\r"
        JSR     PUT_STRING
        HEX     A0 A0 CC CF C4 C5 A0 D2 D5 CE CE C5 D2 A0 C2 CF
        HEX     C1 D2 C4 A0 C5 C4 C9 D4 CF D2 8D AD AD AD AD AD
        HEX     AD AD AD AD AD AD AD AD AD AD AD AD AD AD AD AD
        HEX     AD AD AD AD AD AD AD 8D A0 A0 BC C5 D3 C3 BE A0
        HEX     C1 C2 CF D2 D4 D3 A0 C1 CE D9 A0 C3 CF CD CD C1
```

```
    HEX     CE C4 8D 00

EDITOR_COMMAND_LOOP:
    LDA     GAME_ROWNUM
    CMP     #$09
    BCS     START_LEVEL_EDITOR

    ; "\r"
    ; "COMMAND>"
    JSR     PUT_STRING
    HEX     8D C3 CF CD CD C1 CE C4 BE 00

    JSR     EDITOR_WAIT_FOR_KEY
    LDX     #$00

.loop2:
    LDY     EDITOR_KEYS,X
    BEQ     EDITOR_COMMAND_LOOP_BEEP
    CMP     EDITOR_KEYS,X
    BEQ     .end
    INX
    BNE     .loop2

EDITOR_COMMAND_LOOP_BEEP:
    JSR     BEEP
    JMP     EDITOR_COMMAND_LOOP

.end:
    TXA
    ASL
    TAX
    LDA     EDITOR_ROUTINE_ADDRESS+1,X
    PHA
    LDA     EDITOR_ROUTINE_ADDRESS,X
    PHA
    RTS
```

Defines:
EDITOR_COMMAND_LOOP, used in chunks 72, 74, 241, 244, 261a, and 262.
LEVEL_EDITOR, used in chunk 133a.
START_LEVEL_EDITOR, used in chunks 228, 238, 244, and 268.
Uses BEEP 56, CLEAR_HGR1 5, DRAW_PAGE 45, EDITOR_KEYS 258, EDITOR_ROUTINE_ADDRESS 258, EDITOR_WAIT_FOR_KEY 72, GAME_COLNUM 34a, GAME_MODE 106a, GAME_ROWNUM 34a, INDIRECT_TARGET 245a, INPUT_MODE 66, LIVES 52, PUT_STRING 47, SCORE 50b, and TXTPAGE1 130a.

Clearing a level involves getting the target level number from the user, waiting for the user to insert a valid data disk, and then writing zeros to the target level on disk.

261a ⟨*editor clear level* 261a⟩≡ (278)

```
      ORG     $7C8E
  EDITOR_CLEAR_LEVEL:
      SUBROUTINE

      ; "\r"
      ; ">>CLEAR LEVEL"
      JSR     PUT_STRING
      HEX     8D BE BE C3 CC C5 C1 D2 A0 CC C5 D6 C5 CC 00

      JSR     GET_LEVEL_FROM_KEYBOARD
      BCS     .beep
      JSR     CHECK_FOR_VALID_DATA_DISK

      LDY     #$00
      TYA
  .loop:
      STA     DISK_BUFFER,Y
      INY
      BNE     .loop

      LDA     #DISK_ACCESS_WRITE
      JSR     ACCESS_COMPRESSED_LEVEL_DATA      ; write level
      JMP     EDITOR_COMMAND_LOOP

  .beep:
      JMP     EDITOR_COMMAND_LOOP_BEEP
```

Defines:
EDITOR_CLEAR_LEVEL, used in chunk 258.
Uses CHECK_FOR_VALID_DATA_DISK 238, EDITOR_COMMAND_LOOP 259, GET_LEVEL_FROM_KEYBOARD 74, and PUT_STRING 47.

Moving a level involves getting the source and target level numbers from the user, waiting for the user to insert the source data disk, reading the source level, waiting for the user to insert the target data disk, and then writing the current level data to the target level on disk.

261b ⟨*defines* 4⟩+≡ (281) ◁245c 274c▷

```
      ORG     $824F
  EDITOR_LEVEL_ENTRY:
      HEX     0F
```

Defines:
EDITOR_LEVEL_ENTRY, used in chunk 262.

262 ⟨*editor move level* 262⟩≡ (278)

```
      ORG     $7CD8
  EDITOR_MOVE_LEVEL:
      SUBROUTINE

      ; "\r"
      ; ">>MOVE LEVEL"
      JSR     PUT_STRING
      HEX     8D BE BE CD CF D6 C5 A0 CC C5 D6 C5 CC 00

      JSR     GET_LEVEL_FROM_KEYBOARD
      BCS     .beep
      STY     EDITOR_LEVEL_ENTRY      ; source level

      ; " TO LEVEL"
      JSR     PUT_STRING
      HEX     A0 D4 CF A0 CC C5 D6 C5 CC 00

      JSR     GET_LEVEL_FROM_KEYBOARD
      BCS     .beep
      STY     SAVED_VTOC_DATA         ; convenient place for target level

      ; "\r"
      ; "  SOURCE DISKETTE"
      JSR     PUT_STRING
      HEX     8D A0 A0 D3 CF D5 D2 C3 C5 A0 C4 C9 D3 CB C5 D4 D4 C5 00

      JSR     EDITOR_WAIT_FOR_KEY
      ; Deny and dump user back to editor if not valid data disk
      JSR     CHECK_FOR_VALID_DATA_DISK
      LDA     EDITOR_LEVEL_ENTRY              ; source level
      STA     DISK_LEVEL_LOC
      LDA     #DISK_ACCESS_READ
      JSR     ACCESS_COMPRESSED_LEVEL_DATA      ; read source level

      ; "\r"
      ; "  DESTINATION DISKETTE"
      JSR     PUT_STRING
      HEX     8D A0 A0 C4 C5 D3 D4 C9 CE C1 D4 C9 CF CE A0 C4 C9 D3 CB C5 D4 D4 C5 00

      JSR     EDITOR_WAIT_FOR_KEY
      ; Deny and dump user back to editor if not valid data disk
      JSR     CHECK_FOR_VALID_DATA_DISK
      LDA     SAVED_VTOC_DATA                 ; target level
      STA     DISK_LEVEL_LOC
      LDA     #DISK_ACCESS_WRITE
      JSR     ACCESS_COMPRESSED_LEVEL_DATA      ; write target level
      JMP     EDITOR_COMMAND_LOOP

  .beep:
```

```
    JMP     EDITOR_COMMAND_LOOP_BEEP
```

Defines:
EDITOR_MOVE_LEVEL, used in chunk 258.
Uses CHECK_FOR_VALID_DATA_DISK 238, EDITOR_COMMAND_LOOP 259, EDITOR_LEVEL_ENTRY 261b, EDITOR_WAIT_FOR_KEY 72, GET_LEVEL_FROM_KEYBOARD 74, PUT_STRING 47, and SAVED_VTOC_DATA 240.

263 ⟨*editor play level* 263⟩≡ (278)

```
    ORG     $7C60
EDITOR_PLAY_LEVEL:
    SUBROUTINE

    ; "\r"
    ; ">>PLAY LEVEL"
    JSR     PUT_STRING
    HEX     8D BE BE D0 CC C1 D9 A0 CC C5 D6 C5 CC 00

    JSR     GET_LEVEL_FROM_KEYBOARD
    BCS     .beep

READ_SAVED_INPUT_MODE:
    LDA     #$00
    STA     INPUT_MODE
    LDA     #GAME_MODE_PLAY_IN_EDITOR
    STA     GAME_MODE
    LDA     #$01
    STA     $9D
    LDA     DISK_LEVEL_LOC
    BEQ     .init_game_data_
    LSR     $9D

.init_game_data_:
    JMP     INIT_GAME_DATA

.beep:
    JMP     EDITOR_COMMAND_LOOP_BEEP
```

Defines:
EDITOR_PLAY_LEVEL, used in chunk 258.
Uses GAME_MODE 106a, GET_LEVEL_FROM_KEYBOARD 74, INPUT_MODE 66, and PUT_STRING 47.

264 ⟨*editor edit level* 264⟩≡ (278) 265▷

```
      ORG     $7CBC
  EDITOR_EDIT_LEVEL:
      SUBROUTINE

      ; "\r"
      ; ">>EDIT LEVEL"
      JSR     PUT_STRING
      HEX     8D BE BE C5 C4 C9 D4 A0 CC C5 D6 C5 CC 00

      JSR     GET_LEVEL_FROM_KEYBOARD
      BCS     .beep
      JMP     EDIT_LEVEL

  .beep:
      JMP     EDITOR_COMMAND_LOOP_BEEP
```

Defines:
EDITOR_EDIT_LEVEL, used in chunk 258.
Uses EDIT_LEVEL 265, GET_LEVEL_FROM_KEYBOARD 74, and PUT_STRING 47.

265 ⟨*editor edit level* 264⟩+≡ (278) ◁264

```
        ORG     $7F01
    EDIT_LEVEL:
        SUBROUTINE

        JSR     CLEAR_HGR2
        LDA     #$40
        STA     DRAW_PAGE
        JSR     PUT_STATUS_DRAW
        LDA     #$20
        STA     DRAW_PAGE

        JSR     CHECK_FOR_VALID_DATA_DISK
        LDX     #$01
        STX     $AD
        DEX
        JSR     LOAD_LEVEL
        BCC     .start_editing
        JMP     EDITOR_COMMAND_LOOP_BEEP

    .start_editing:
        LDA     #$00
        STA     GAME_COLNUM
        STA     GAME_ROWNUM

    EDIT_LEVEL_KEY_LOOP:
        JSR     GET_KEY_FOR_EDIT_LEVEL
        CMP     #$BA
        BCS     .store_sprite_num       ; key >= '9'+1
        CMP     #$B0
        BCC     .store_sprite_num       ; key < '0'
        ; key is digit

        AND     #$0F
        STA     SPRITE_NUM
        LDY     GAME_ROWNUM
        ⟨set active row pointer PTR1 for Y 78c⟩
        LDY     GAME_COLNUM
        LDA     SPRITE_NUM
        EOR     (PTR1),Y
        BEQ     .store_sprite
        LSR     $AD

    .store_sprite:
        LDA     SPRITE_NUM
        STA     (PTR1),Y
        JSR     DRAW_SPRITE_PAGE1
        JMP     EDIT_LEVEL_KEY_LOOP

    .store_sprite_num:
```

```
    STA     SPRITE_NUM
    LDY     #$FF

.loop2:
    INY
    LDA     LEVEL_EDIT_KEY_TABLE,Y
    BEQ     EDIT_LEVEL_BEEP

    CMP     SPRITE_NUM
    BNE     .loop2

    TYA
    ASL
    TAY
    LDA     LEVEL_EDIT_KEY_FUNCTIONS+1,Y
    PHA
    LDA     LEVEL_EDIT_KEY_FUNCTIONS,Y
    PHA
    RTS

EDIT_LEVEL_BEEP:
    JSR     BEEP
    JMP     EDIT_LEVEL_KEY_LOOP
```

Defines:
EDIT_LEVEL, used in chunks 264 and 268.
Uses BEEP 56, CHECK_FOR_VALID_DATA_DISK 238, CLEAR_HGR2 5, DRAW_PAGE 45, DRAW_SPRITE_PAGE1 35, GAME_COLNUM 34a, GAME_ROWNUM 34a, GET_KEY_FOR_EDIT_LEVEL 266, LEVEL_EDIT_KEY_FUNCTIONS 267, LEVEL_EDIT_KEY_TABLE 267, LOAD_LEVEL 111b, PTR1 78b, and SPRITE_NUM 25c.

266 ⟨*get key for edit level* 266⟩≡ (278)

```
    ORG     $814B
GET_KEY_FOR_EDIT_LEVEL:
    SUBROUTINE

    LDY     GAME_ROWNUM
    ⟨set active row pointer PTR1 for Y 78c⟩
    LDY     GAME_COLNUM
    LDA     (PTR1),Y
    JSR     WAIT_FOR_KEY_WITH_CURSOR_PAGE_1
    STA     KBDSTRB
    RTS
```

Defines:
GET_KEY_FOR_EDIT_LEVEL, used in chunk 265.
Uses GAME_COLNUM 34a, GAME_ROWNUM 34a, KBDSTRB 68b, PTR1 78b, and WAIT_FOR_KEY_WITH_CURSOR_PAGE_1 71.

267 ⟨*tables* 9⟩+≡ (281) ◁258

```
      ORG     $8162
  LEVEL_EDIT_KEY_TABLE:
      ; J (left)
      ; I (up)
      ; K (right)
      ; M (down)
      ; ctrl-S (save)
      ; right arrow
      ; left arrow
      ; ctrl-Q
      HEX     CA C9 CB CD 93 95 88 91 00
  LEVEL_EDIT_KEY_FUNCTIONS:
      WORD    LEVEL_EDIT_LEFT-1
      WORD    LEVEL_EDIT_UP-1
      WORD    LEVEL_EDIT_RIGHT-1
      WORD    LEVEL_EDIT_DOWN-1
      WORD    LEVEL_EDIT_SAVE-1
      WORD    LEVEL_EDIT_RIGHT_ARROW-1
      WORD    LEVEL_EDIT_LEFT_ARROW-1
      WORD    LEVEL_EDIT_Q-1
```

Defines:
LEVEL_EDIT_KEY_FUNCTIONS, used in chunk 265.
LEVEL_EDIT_KEY_TABLE, used in chunk 265.

268 ⟨*level editor key functions* 268⟩≡ (278)

```
      ORG     $7F74
  LEVEL_EDIT_UP:
      SUBROUTINE

      LDA     GAME_ROWNUM
      BEQ     EDIT_LEVEL_BEEP         ; nope!

      DEC     GAME_ROWNUM
      BPL     EDIT_LEVEL_KEY_LOOP     ; unconditional

  LEVEL_EDIT_LEFT:
      SUBROUTINE

      LDA     GAME_COLNUM
      BEQ     EDIT_LEVEL_BEEP         ; nope!

      DEC     GAME_COLNUM
      BPL     EDIT_LEVEL_KEY_LOOP     ; unconditional

  LEVEL_EDIT_RIGHT:
      SUBROUTINE

      LDA     GAME_COLNUM
      CMP     #MAX_GAME_COL
      BCS     EDIT_LEVEL_BEEP         ; nope!

      INC     GAME_COLNUM
      BNE     EDIT_LEVEL_KEY_LOOP     ; unconditional

  LEVEL_EDIT_DOWN:
      SUBROUTINE

      LDA     GAME_ROWNUM
      CMP     #MAX_GAME_ROW
      BCS     EDIT_LEVEL_BEEP         ; nope!

      INC     GAME_ROWNUM
      BNE     EDIT_LEVEL_KEY_LOOP     ; unconditional

  LEVEL_EDIT_SAVE1:
      SUBROUTINE

      LDA     GAME_ROWNUM
      PHA
      LDA     GAME_COLNUM
      PHA
      LDA     #$01
      JSR     ACCESS_HI_SCORE_DATA_FROM_DISK
      CMP     #$00
```

```
    BNE     .check_for_master_disk

    JSR     BAD_DATA_DISK
    JMP     .end

.check_for_master_disk:
    CMP     #$01
    BNE     .save_data

    JSR     DONT_MANIPULATE_MASTER_DISK
    JMP     .end

.save_data:
    JSR     COMPRESS_AND_SAVE_LEVEL_DATA
    PLA
    STA     GAME_COLNUM
    PLA
    STA     GAME_ROWNUM
    LDA     #$01
    STA     $AD
    RTS

.end:
    LDA     #$00
    STA     GAME_COLNUM
    STA     GAME_ROWNUM
    JMP     EDIT_LEVEL_KEY_LOOP

LEVEL_EDIT_SAVE:
    SUBROUTINE

    JSR     LEVEL_EDIT_SAVE1
    JMP     EDIT_LEVEL_KEY_LOOP

LEVEL_EDIT_RIGHT_ARROW:
    SUBROUTINE

    LDA     DISK_LEVEL_LOC
    CMP     #$95

LEVEL_EDIT_RIGHT_ARROW_CHECK:
    BEQ     EDIT_LEVEL_BEEP
    JSR     LEVEL_EDIT_CHANGE_LEVEL
    INC     DISK_LEVEL_LOC
    INC     LEVELNUM
    JMP     EDIT_LEVEL

LEVEL_EDIT_LEFT_ARROW:
    SUBROUTINE
```

```
    LDA     DISK_LEVEL_LOC
    BEQ     LEVEL_EDIT_RIGHT_ARROW_CHECK
    JSR     LEVEL_EDIT_CHANGE_LEVEL
    DEC     LEVELNUM
    DEC     DISK_LEVEL_LOC
    JMP     EDIT_LEVEL

LEVEL_EDIT_Q:
    SUBROUTINE

    JSR     LEVEL_EDIT_CHANGE_LEVEL
    JMP     START_LEVEL_EDITOR

LEVEL_EDIT_CHANGE_LEVEL:
    SUBROUTINE

    LDA     $AD
    BNE     .end
    JSR     CLEAR_HGR2
    LDA     #$40
    STA     DRAW_PAGE
    LDA     #$00
    STA     GAME_COLNUM
    STA     GAME_ROWNUM

    ; "LEVEL HAS BEEN CHANGED BUT\r"
    ; "NOT SAVED. DO YOU WISH TO\r"
    ; "SAVE MODIFIED LEVEL (Y/N) "
    JSR     PUT_STRING
    HEX     CC C5 D6 C5 CC A0 C8 C1 D3 A0 C2 C5 C5 CE A0 C3
    HEX     C8 C1 CE C7 C5 C4 A0 C2 D5 D4 8D CE CF D4 A0 D3
    HEX     C1 D6 C5 C4 AE A0 C4 CF A0 D9 CF D5 A0 D7 C9 D3
    HEX     C8 A0 D4 CF 8D D3 C1 D6 C5 A0 CD CF C4 C9 C6 C9
    HEX     C5 C4 A0 CC C5 D6 C5 CC A0 A8 D9 AF CE A9 A0 00

    JSR     BEEP
    STA     TXTPAGE2

.loop:
    LDA     #$00
    JSR     WAIT_FOR_KEY
    STA     KBDSTRB
    CMP     #$CE        ; 'N'
    BEQ     .end

    CMP     #$D9        ; 'Y'
    BNE     .loop

    JSR     LEVEL_EDIT_SAVE1
```

```
.end:
    STA     TXTPAGE1
    LDA     #$00
    STA     GAME_COLNUM
    STA     GAME_ROWNUM
    RTS
```

Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231, BAD_DATA_DISK 237a, BEEP 56, CLEAR_HGR2 5, COMPRESS_AND_SAVE_LEVEL_DATA 116, DONT_MANIPULATE_MASTER_DISK 237b, DRAW_PAGE 45, EDIT_LEVEL 265, GAME_COLNUM 34a, GAME_ROWNUM 34a, KBDSTRB 68b, LEVELNUM 52, PUT_STRING 47, START_LEVEL_EDITOR 259, TXTPAGE1 130a, TXTPAGE2 122c, and WAIT_FOR_KEY 70.

Chapter 12

Extra data

From 9000 through 9EFF appears to be unused data. 9E00-9EFF is interesting because it seems to contain a fragment of source code:

```
   QU $1C35
ytable EQU $1C51
bytable EQU $1C62
bitable EQU $1C7E
xbytable EQU $1C9A
xbitable EQU $1D26
boot EQU $1DB2
scorebuf EQU $1F00
chardata EQU $AD00

rwtsparm EQU $B7E8
rwtsvolm EQU $B7EB
rwtstrck EQU $B7EC
rwtssect EQU $B7ED
rwtsbuff EQU $B7F0
rwtscmn
```

Chapter 13

The whole thing

We then put together the entire assembly file, including all the data that happened to be in the uninitialized sections.

273 ⟨*zero page initial* 273⟩≡ (278)

```
      ORG     $0000
  ZERO_PAGE_INITIAL:
      HEX     4C 00 28 00 C0 68 0D 00 60 10 00 40 D0 3F 00 10
      HEX     D4 B4 9C FA B0 FF 37 FB FE FE FC EC 76 D7 E6 20
      HEX     00 28 00 4C 00 00 00 D5 00 04 18 60 20 0F 0C D6
      HEX     FC FA FF 73 FD 8D 50 8E B5 B7 04 FF FB B7 00 1F
      HEX     00 18 01 60 00 00 14 D8 E8 B7 91 9A 04 02 00 08
      HEX     00 FD 1C 3C EC AC B4 38 C4 E9 B6 FF BA BB 2A FE
      HEX     86 3E 85 3A A6 3E 86 40 A0 00 A5 3A 84 3C 85 3D
      HEX     A6 08 20 AE 00 C5 00 D0 F9 20 AE 00 C5 01 D0 F5
      HEX     20 AE 00 C5 02 D0 01 20 8C C0 10 FB 06 85 3F BD
      HEX     8C C0 10 FB 25 CA 00 3C C8 FF EC 0E 00 C0 BD 8C
      HEX     C0 10 FB C5 03 D0 BD 00 3D C6 40 D0 DA 60 BD 8C
      HEX     C0 10 FB 60 A2 D4 86 00 E8 86 01 E8 86 02 E8 86
      HEX     03 A9 04 AA 60 FF 2A 29 B1 FF 48 2A 09 01 8A 49
      HEX     FF C9 10 AA E8 09 AA EA FF A2 00 48 CA D0 FC A2
      HEX     0F BD F0 04 9D 00 01 CA D0 F7 9A 60 18 69 05 29
      HEX     07 69 01 00 EA E8 CA E8 CA E8 CA C8 88 C8 88 C8
```

274a ⟨*stack initial* 274a⟩≡ (278)

```
        ORG     $0100
    STACK_INITIAL:
        HEX     BE B9 B7 B4 7B 63 90 61 FF 8B FE 07 FF 03 FF 5F
        HEX     5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 7A
        HEX     5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 7A
        HEX     5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 7A
        HEX     5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 7A
        HEX     5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 5A 7A
        HEX     1D 7C FE 31 29 12 2D 60 FB 5D F7 72 D9 ED D1 A8
        HEX     FC FC FE 7B F7 FF 5E FF 35 FE EC CD BF EF 3D DD
        HEX     DF EF FF DF F7 7B 7F BF 7F BF ED E5 BF DF ED D4
        HEX     01 2E 00 10 4A 84 48 83 02 22 50 82 9A 0C 46 04
        HEX     D7 19 AD 7F 4C C7 DF BE FF 8F 7B CF DF FF B3 DA
        HEX     00 08 02 F2 00 05 86 00 09 F3 00 63 08 00 43 02
        HEX     7F DF FD CF EB BF AF BF D5 FF C7 E7 BD FF BF 7B
        HEX     C9 E4 61 06 19 07 04 56 01 63 42 0F 93 43 C2 46
        HEX     89 4A 00 52 05 2F 22 A3 A0 37 00 84 0A 01 80 40
        HEX     00 09 00 06 17 26 FC 00 26 FC 6F BE 66 BE C6 42
```

274b ⟨*random initial* 274b⟩≡ (278)

```
        ORG     $0200
    RANDOM_INIT_DATA:
        INCLUDE "random_init_data.asm"

        ORG     $0C00
    MORE_RANDOM_INIT_DATA:
        INCLUDE "more_random_init_data.asm"

        ORG     $1EB2
    PADDING:
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Upon load, the Lode Runner file on disk gets dumped into memory at `0800-88FF` and then is jumped to at `0800`. The routine there, `RELOCATE0`, relocates the entire file to `3F00-BFFF` and then jumps to `RELOCATE1`. That routine can relocate various segments, but in this case it only moves `3F00-5EFF` to `0000-1FFF` and leaves the rest alone.

There is some `UNUSED` code after `RELOCATE1` which appears to zero out most (but not all) of graphics page 1. It seems to leave the first `0xA8` bytes untouched.

274c ⟨*defines* 4⟩+≡ (281) ◁261b

```
    SRC_PTR     EQU     $3C     ; 2 bytes
    DEST_PTR    EQU     $3E     ; 2 bytes
```

275 ⟨*relocation routine* 275⟩≡ (281)

```
        ORG     $5F00
    RELOCATE0_ALIAS:
        SUBROUTINE

        LDA     #$64
        STA     $0800
        LDA     #$76
        STA     $0801
        LDA     #$0D
        STA     $0802

        ; Copy 0x8100 bytes from $0800-$88FF to $3F00-$BFFF
        LDY     #$88
        BEQ     RELOCATE1       ; Never happens
        STY     SRC_PTR+1
        LDA     #$BF
        STA     DEST_PTR+1
        LDX     #$81
        LDY     #$00
        STY     SRC_PTR
        STY     DEST_PTR

    .loop:
        LDA     (SRC_PTR),Y
        STA     (DEST_PTR),Y
        INY
        BNE     .loop
        DEC     SRC_PTR+1
        DEC     DEST_PTR+1
        DEX
        BNE     .loop
        JMP     RELOCATE1

    RELOCATE1:
        SUBROUTINE

        LDX     #<RELOCATE_TABLE

    .loop:
        LDA     RELOCATE_TABLE-<RELOCATE_TABLE,X
        STA     .rd_insn+2
        INX
        LDA     RELOCATE_TABLE-<RELOCATE_TABLE,X
        STA     .cmp_insn+1
        INX
        LDA     RELOCATE_TABLE-<RELOCATE_TABLE,X
        STA     .wr_insn+2
        INX
        LDY     #$00
```

```
.loop2:
.rd_insn:
    LDA     $0800,Y     ; address replaced above
.wr_insn:
    STA     $0800,Y     ; address replaced above
    INY
    BNE     .loop2

    INC     .rd_insn+2
    INC     .wr_insn+2
    LDA     .rd_insn+2
.cmp_insn:
    CMP     #$FF        ; comparison value replaced above
    BNE     .loop2

    DEC     RELOCATE_SEGMENT_COUNT
    BNE     .loop

    JMP     MAIN

UNUSED:
    SUBROUTINE

    LDX     #$A8

.loop:
    LDA     #$00

.loop2:
.wr_insn:
    STA     $2000,X
    INX
    BNE     .loop2
    INC     .wr_insn+2
    LDA     .wr_insn+2
    CMP     #$40
    BNE     .loop

    ; Fall through to MAIN

    ORG     $5FA6
RELOCATE_SEGMENT_COUNT:
    HEX     04
RELOCATE_TABLE:
    HEX     3F 47 00
    HEX     47 5F 08
    HEX     60 8E 60
    HEX     8E C0 8E
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Defines:
RELOCATE0_ALIAS, never used.
RELOCATE1, never used.
Uses MAIN 130b.

Dead code is code that disassembles, but doesn't seem to be called.

277a ⟨*dead code* 116⟩+≡ (281) ◁141a

```
        ORG     $6000

        JSR     DETECT_LACK_OF_JOYSTICK
        LDA     #$01
        JSR     ACCESS_HI_SCORE_DATA_FROM_DISK

        ; Fallthrough to RESET_GAME
```

Uses ACCESS_HI_SCORE_DATA_FROM_DISK 231 and DETECT_LACK_OF_JOYSTICK 68a.

277b ⟨*zeroed areas* 277b⟩≡ (281)

```
        ORG     $8D53
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        HEX     00 00 00 00 00 00 00 00 00 00 00 00 00

        ORG     $8E46
        HEX     00 00 00 00 00 00 00 00 00 00

        ORG     $8E53
        ; through to $8FFF.
        DS      $1AD
```

277c ⟨*garbage area* 277c⟩≡ (281)

```
        ORG     $9000
        INCLUDE "garbage.asm"

        ORG     $B5F0
        HEX     C5 A2 D4 A2 00 D0 8A 94 80 80 80 80 80 80 80 80
```

277d ⟨*dos* 277d⟩≡ (281)

```
        ORG     $B600
        INCLUDE "dos.asm"
```

278 ⟨*routines* 5⟩+≡ (281) ◁90

```
; Initialized "uninitialized" data

⟨zero page initial 273⟩
⟨stack initial 274a⟩
⟨random initial 274b⟩

; Sprite routines

⟨erase sprite at screen coordinate 38⟩
⟨draw sprite at screen coordinate 41⟩
⟨draw player 43⟩
⟨char to sprite num 44⟩
⟨put char 46a⟩
⟨put string 47⟩
⟨put digit 48a⟩
⟨to decimal3 49⟩
⟨bcd to decimal2 50a⟩

; Screen and level routines

⟨add and update score 51⟩
⟨put status 53⟩
⟨level draw routine 77⟩
⟨set active and background row pointers PTR1 and PTR2 for Y routine 79c⟩
⟨splash screen 124⟩
⟨construct and display high score screen 119b⟩
⟨iris wipe 89⟩
⟨iris wipe step 92⟩
⟨draw wipe step 94a⟩
⟨draw wipe block 98⟩
⟨load compressed level data 108⟩
⟨load level 111b⟩

; Sound routines

⟨beep 56⟩
⟨load sound data 58⟩
⟨append note 59a⟩
⟨play note 60a⟩
⟨sound delay 61a⟩
⟨play sound 62⟩
⟨append level cleared note 63⟩

; Joystick routines

⟨read paddles 65⟩
⟨check joystick or delay 67⟩
⟨detect lack of joystick 68a⟩
```

```
; Keyboard routines
```

⟨*wait key* 69a⟩
⟨*wait key queued* 69b⟩
⟨*wait for key* 70⟩
⟨*wait for key page1* 71⟩
⟨*editor wait for key* 72⟩
⟨*hit key to continue* 73a⟩
⟨*get level from keyboard* 74⟩

```
; Player movement routines
```

⟨*get player sprite and coord data* 135b⟩
⟨*increment player animation state* 136a⟩
⟨*check for gold picked up by player* 137⟩
⟨*check for input* 139⟩
⟨*ctrl handlers* 140a⟩
⟨*return handler* 145⟩
⟨*check buttons* 151⟩
⟨*try moving up* 154⟩
⟨*try moving down* 159⟩
⟨*try moving left* 161⟩
⟨*try moving right* 164⟩
⟨*try digging left* 168⟩
⟨*try digging right* 171⟩
⟨*drop player in hole* 174⟩
⟨*move player* 175⟩
⟨*check for mode 1 input* 148⟩

```
; Guard AI routines
```

⟨*guard resurrections* 188⟩
⟨*guard store and load data* 186⟩
⟨*get guard sprite and coords* 187b⟩
⟨*move guards* 191a⟩
⟨*move guard* 193⟩
⟨*determine guard move* 198⟩
⟨*determine guard left right limits* 201⟩
⟨*should guard move left* 215⟩
⟨*should guard move right* 217⟩
⟨*should guard move up or down* 219⟩
⟨*nudge guards* 183⟩
⟨*check for gold picked up by guard* 184⟩
⟨*increment guard animation state* 185⟩
⟨*try guard move left* 204⟩
⟨*try guard move right* 206⟩
⟨*try guard move up* 208⟩
⟨*try guard move down* 210⟩
⟨*pseudo distance* 213⟩
⟨*guard drop gold* 212⟩

⟨*guard find candidate row below* 221⟩
⟨*guard find candidate row above* 224⟩

```
; Disk routines
```

⟨*rwts targets* 245a⟩
⟨*jump to RWTS indirectly* 106b⟩
⟨*indirect call* 245b⟩
⟨*bad data disk* 237a⟩
⟨*dont manipulate master disk* 237b⟩
⟨*access hi score data* 231⟩
⟨*record hi score data* 233⟩
⟨*check for valid data disk* 238⟩
⟨*editor initialize disk* 241⟩
⟨*editor clear high scores* 244⟩
⟨*format patch* 228⟩

```
; Startup code
```

⟨*startup code* 129⟩
⟨*check for button down* 131b⟩
⟨*no button pressed* 132a⟩
⟨*button pressed at startup* 132b⟩
⟨*key pressed at startup* 132c⟩
⟨*ctrl-e pressed* 133a⟩
⟨*return pressed* 133b⟩
⟨*timed out waiting for button or keypress* 133c⟩
⟨*check game mode* 134a⟩
⟨*reset game if not mode 1* 134b⟩
⟨*display high score screen* 134c⟩
⟨*long delay attract mode* 134d⟩

```
; Game loop
```

⟨*Initialize game data* 246⟩
⟨*game loop* 250⟩
⟨*handle timers* 126⟩
⟨*do ladders* 179⟩
⟨*anims* 254⟩
⟨*show anim line* 256⟩

```
; Editor routines
```

⟨*level editor* 259⟩
⟨*editor clear level* 261a⟩
⟨*editor move level* 262⟩
⟨*editor play level* 263⟩
⟨*editor edit level* 264⟩
⟨*get key for edit level* 266⟩
⟨*level editor key functions* 268⟩

281 ⟨* 281⟩≡

```
PROCESSOR 6502
⟨defines 4⟩
⟨tables 9⟩
⟨routines 5⟩
⟨relocation routine 275⟩
⟨dead code 116⟩
⟨zeroed areas 277b⟩
⟨garbage area 277c⟩
⟨dos 277d⟩
```

Chapter 14

Defined Chunks

⟨ * 281⟩ 281
⟨ROW_ADDR = $9E00 + LEVELNUM * $0100 109b⟩ 109a, 109b
⟨WIPE0 = WIPE_COUNTER 100a⟩ 92, 100a
⟨WIPE1 = 0 100b⟩ 92, 100b
⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 101d⟩ 92, 101d
⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 103a⟩ 93, 103a
⟨WIPE2 += 4 * WIPE1 + 6 102⟩ 93, 102
⟨WIPE2 = 2 * WIPE0 100c⟩ 92, 100c
⟨WIPE2 = 3 - WIPE2 100d⟩ 92, 100d
⟨WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER 100e⟩ 92, 100e
⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 100f⟩ 92, 100f
⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 101a⟩ 92, 101a
⟨WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7 101b⟩ 92, 101b
⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 101c⟩ 92, 101c
⟨*access hi score data* 231⟩ 231, 278
⟨*add and update score* 51⟩ 51, 278
⟨*anims* 254⟩ 254, 278
⟨*append level cleared note* 63⟩ 63, 278
⟨*append note* 59a⟩ 59a, 278
⟨*bad data disk* 237a⟩ 237a, 278
⟨*bcd to decimal2* 50a⟩ 50a, 278
⟨*beep* 56⟩ 56, 278
⟨*button pressed at startup* 132b⟩ 132b, 278
⟨*char to sprite num* 44⟩ 44, 278
⟨*check buttons* 151⟩ 151, 278
⟨*check for button down* 131b⟩ 131b, 278
⟨*check for gold picked up by guard* 184⟩ 184, 278
⟨*check for gold picked up by player* 137⟩ 137, 278
⟨*check for input* 139⟩ 139, 278
⟨*check for mode 1 input* 148⟩ 148, 278
⟨*check for valid data disk* 238⟩ 238, 278

⟨*check game mode* 134a⟩ 134a, 278
⟨*check joystick or delay* 67⟩ 67, 278
⟨*construct and display high score screen* 119b⟩ 119b, 278
⟨*Copy data from* ROW_ADDR *into* DISK_BUFFER 109c⟩ 109a, 109c
⟨*Copy level data* 109a⟩ 108, 109a
⟨*ctrl handlers* 140a⟩ 140a, 140b, 141b, 141c, 142a, 142b, 143, 144a, 278
⟨*ctrl-e pressed* 133a⟩ 133a, 278
⟨*dead code* 116⟩ 116, 141a, 277a, 281
⟨*Decrement* WIPE0 103b⟩ 93, 103b
⟨*Decrement* WIPE10 *modulo 7* 104b⟩ 93, 104b
⟨*Decrement* WIPE4 105a⟩ 93, 105a
⟨*Decrement* WIPE6 104d⟩ 93, 104d
⟨*Decrement* WIPE8 *modulo 7* 105c⟩ 93, 105c
⟨*defines* 4⟩ 4, 22, 23, 25c, 28b, 29b, 34a, 40, 45, 46b, 48b, 50b, 52, 57, 59b, 61b, 64, 66, 68b, 69c, 78b, 80c, 81d, 81e, 84b, 88, 91, 106a, 107, 111a, 112a, 118, 122c, 130a, 136b, 138a, 144b, 166, 181, 190, 192a, 197, 200, 220, 227, 229, 232, 239, 245c, 261b, 274c, 281
⟨*detect lack of joystick* 68a⟩ 68a, 278
⟨*determine guard left right limits* 201⟩ 201, 278
⟨*determine guard move* 198⟩ 198, 278
⟨*display high score screen* 134c⟩ 134c, 278
⟨*do ladders* 179⟩ 179, 278
⟨*dont manipulate master disk* 237b⟩ 237b, 278
⟨*dos* 277d⟩ 277d, 281
⟨*draw high score* 122a⟩ 120b, 122a
⟨*draw high score initials* 121b⟩ 120b, 121b
⟨*draw high score level* 121c⟩ 120b, 121c
⟨*draw high score row number* 120c⟩ 120b, 120c
⟨*draw high score rows* 120b⟩ 119b, 120b
⟨*draw high score table header* 120a⟩ 119b, 120a
⟨*draw player* 43⟩ 43, 278
⟨*draw sprite at screen coordinate* 41⟩ 41, 278
⟨*draw wipe block* 98⟩ 98, 278
⟨*Draw wipe for north part* 95⟩ 94a, 95
⟨*Draw wipe for north2 part* 96⟩ 94a, 96
⟨*Draw wipe for south part* 94b⟩ 94a, 94b
⟨*Draw wipe for south2 part* 97⟩ 94a, 97
⟨*draw wipe step* 94a⟩ 94a, 278
⟨*drop player in hole* 174⟩ 174, 278
⟨*editor clear high scores* 244⟩ 244, 278
⟨*editor clear level* 261a⟩ 261a, 278
⟨*editor edit level* 264⟩ 264, 265, 278
⟨*editor initialize disk* 241⟩ 241, 278
⟨*editor move level* 262⟩ 262, 278
⟨*editor play level* 263⟩ 263, 278
⟨*editor wait for key* 72⟩ 72, 278

Chapter 15

Index