The ALPACA Operating System for Z-80 based computers **Draft 0.8**

Scott “Jerry” Lawrence
alpaca@umlautllama.com

August 22, 2003

Contents

Chapter 1

Overview

1.1 This Document

This document describes and implements ALPACA. ALPACA is a multitasking operating system designed for Pac-Man[1] and Pengo[2] arcade hardware.

This document contains the all-original source code (Z-80 ASM) to build the core operating system, as well as a few example tasks. The asm file generated by this document (`alpaca.asm`) is commented as well so this document is not needed to understand what is going on in that file.[3] This document can be used alone or as the reference for the generated .asm file.

Pengo is included as well for the explanations since the basic hardware is identical to Pac-Man, albeit with its control registers and layout of the hardware differing slightly. In fact, Pengo hardware is a superset of Pac-Man hardware. Anything that runs on Pac hardware should run on Pengo. Pengo adds some other hardware, like the ability to switch graphics banks, as well as some extra ram, but those details are outside of the scope of this document.

About the only main differences is that the sound and color PROMS are layed out differently. This will result in colors being "off", or the sound not sounding right.

It should also be noted that all of the graphics used in the graphics roms are completely original to avoid copyright issues with NAMCO, SEGA, or whomever currently holds the copyrights for the original program and graphics code.

[1] Pac-Man is copyright and trademark NAMCO.

[2] Pengo is copyright and trademark SEGA.

[3] I know that this goes against the reason for using noweb, but this is meant to be used as a learning device for others, and I feel that having fully documented asm is important for this purpose.

1.2 Hardware Limitations

The hardware has some distinct and extreme limitations. The most important of these limitations are:

- 1 Kb (1024 bytes) of RAM
- 16 Kb (16384 bytes) of ROM (Pac-Man hardware)
- background of 8x8 tiled characters, four colors each (1 Kb)
- 6 floating sprites (16x16 pixels, four colors) (1 Kb)

Ms. Pac-Man adds another 8Kb (8192 bytes) of non-contiguous ROM.

Pengo hardware doubles the RAM to 2 Kb, and has 36 Kb of contiguous ROM, making for a much more flexible system. Due to the fact that we're writing this for Pac hardware primarilly, we will not exploit these advantages within the kernel of this OS. If we write this for the smaller of the two, then it will work on both.

1.3 Project Goals

The goals of ALPACA are to provide task management, messaging, basic semaphores, simple ram management and a graphical user interface for a few tasks concurrently running on the arcade machine computer. The number of runnable tasks will be fixed. This all comes together to form a fully pre-emptive multitasking operating system can be built on such a tight hardware platform.

I fully realize that there are other multitasking OS's for the Z80 architecture. I know that this is not the first, but I highly doubt any other package is as fully documented as this one.

The design of the architecture is detailed in §2.

The footprint of the OS Kernel is designed to be very small to allow for user code and data to be as large as possible.

Being that the OS is currently in development, I'm shooting for no more than 1Kb (1024 bytes) of space to be used by the kernel, library functions and data, allowing for 15Kb (15360 bytes) of program space for applications and games to be implemented. I'm also trying to keep the number of sprites and tiles used down to a minimum as well for similar reasons. The OS uses upper and lowercase character sprites, but this can always be reduced down to just one or the other to gain back 26 character positions.

Chapter 2

System Architecture

This chapter explains how the kernel and memory of the system are arranged.

2.1 Hardware Architecture

First of all, we'll start with how the hardware is arranged. If you look at figure 2.1, you will see the memory map for Pac-Man based games on the left, and Pengo on the right. Pengo is only really shown as reference since it was mentioned earlier in this doc. All of the design described here will focus on Pac-Man hardware.

In a nutshell, there is some ROM on the system, shown in green. There also are some control registers which allow the program to get input from the user (joystick, coin switches, etc) which are shown in blue. This group also contains things like a flag to flip the screen, as well as the watchdog timer.

The watchdog timer is a device that resets the system completely unless it has been cleared within 16 screen refreshes. This is made for when a game might get into some unpredicted behavior where it might crash or hang. When the game gets to that state, it will reboot itself using this mechanism. We will essentially disable it by clearing it within the interrupt routine which happens once every screen referesh.

2.2 RAM Allocation

There are three groups of RAM, shown in pink in figure 2.1. These are the screen color and character RAM, as well as User RAM. The screen color and character RAM are for drawing things on the screen. The hardware has a character-based background, where you put the character to draw in the character RAM and the color to draw it in the color RAM.

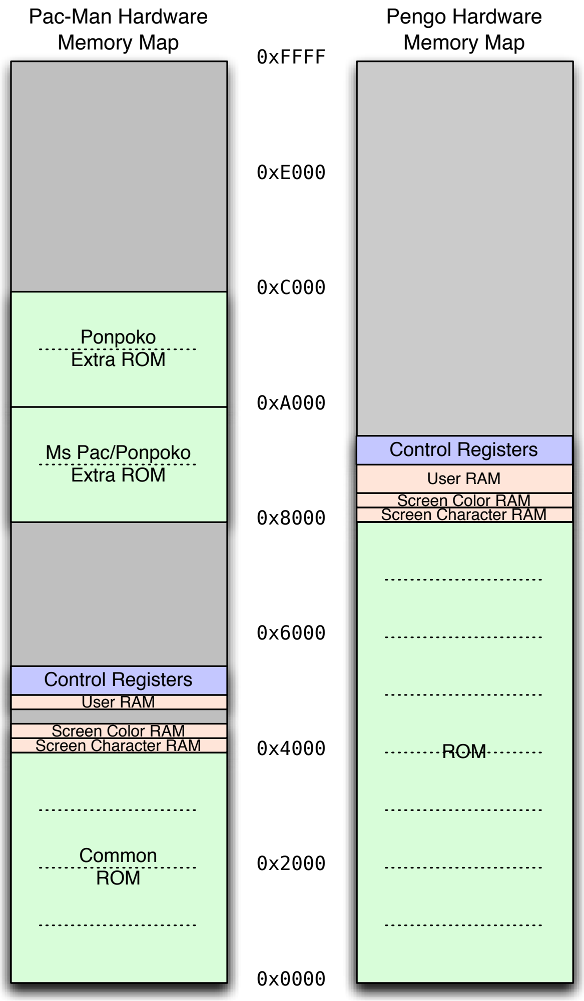

Figure 2.1: Hardware memory map

The other RAM is the User Ram, which is general purpose, for whatever the program/programmer wants to use it for. The exception is the uppermost 16 bytes, which is used to draw floating sprites on the screen.

Figure 2.2 shows just the User Ram on the system. This shows how ALPACA uses the ram. It is broken up into 6 sections. This diagram assumes that there are four tasks concurrently running. More about those in §8.

The sections shown are: (from top to bottom)

- Sprite Ram (16 bytes)
- Task 0 Stack (192 bytes)
- Task 1 Stack (192 bytes)
- Task 2 Stack (192 bytes)
- Task 3 Stack (192 bytes)
- Semaphores (16 bytes)
- Message Queue (64 bytes)
- Kernel and Task Globals (160 bytes)

2.2.1 Sprite Ram

This is a section of RAM that is used by the sprite video hardware. This is where the positions, colors, sprite numbers and flags are placed by the software to have the video hardware draw the sprites on the screen.

2.2.2 Task Stacks

Each task will have its own stack pointer and stack. Figure 2.2 shows four task stacks in the system for up to four tasks running. If we had more ram or a disk for virtual memory, we could probably increase this to be virtually unlimited, but for now, we'll stick to four.

When each task is enabled by the task switcher[1] it needs to be within its own stack frame. Each task thinks that only itself is running. There are some rudimentary communications methods by which one task can talk to another, and that is via the Message Queue, which is discussed next. Other than the Message Queue, the task has no idea if there is one other task, or thirty other tasks running on the system.

[1]See §8 for more information.

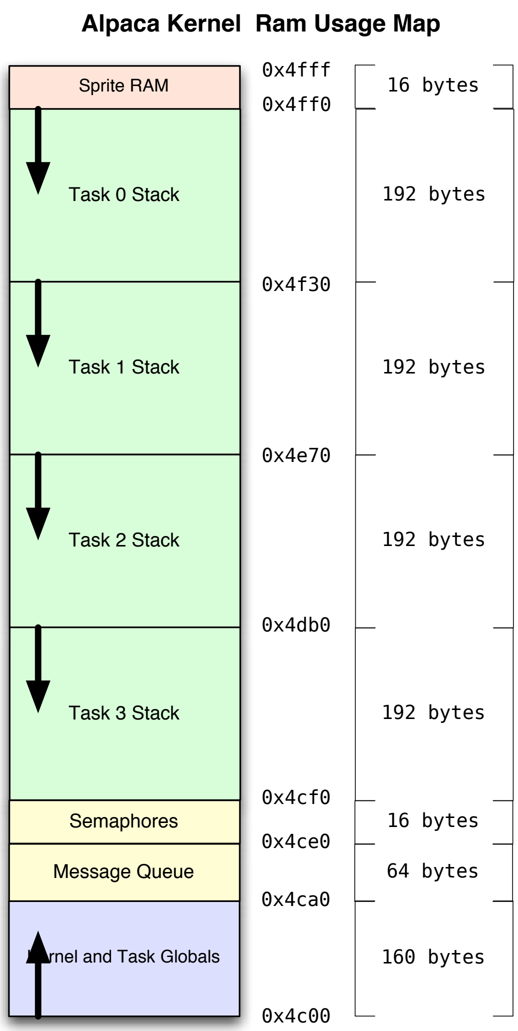

Figure 2.2: Kernel RAM memory map

2.2.3 Semaphores

This is the ram where the kernel will keep track of the state of all of the semaphores that are in use in the system. More about those in §5.

2.2.4 Message Queue

The message queue is a small amount of memory (256 bytes) that contains rudimentary messages (TBD) that allow for a task to communicate with the kernel or with other tasks.

More details about the message queue can be found in §6.

2.2.5 Kernel and Task Globals

This section of memory contains all of the variables used by the kernel itself as well as all of the tasks themselves. Since there is no memory protection at all all of this has to be cooridinated such that multiple tasks are prevented from assuming control of RAM that another task or the kernel is using. Obviously, this cannot be enforced, so it is the obligation of the task to “play nice” with the other tasks, and stay within its own sandbox.

The memory allocation routines are discussed in §7.

Chapter 3

System Initialization

This chapter describes what the system does as it starts up, and how it initializes all of the hardware and software modules.

1. Hardware Initialization - zero all ram
2. Splash Screen Display
3. Initialize Tasks
4. Start Runtime

13a ⟨*.start implementation* 13a⟩≡
```
.start:
⟨start hardware init 13b⟩
⟨start initialize tasks 17b⟩
⟨start enable interrupts 15b⟩
⟨start splash screen 16⟩
```
This code is used in chunk 102.

3.1 Hardware Initialization

This gets called immediately from the `RST 00` call, as defined in §4, which basically is simply a `jp` to here at memory location `0x0000`, which is where execution starts when the processor is turned on.

Okay, so the first thing that happens is that we head over to the `.startup` block, where lots of things will be setup.

13b ⟨*start hardware init* 13b⟩≡
```
        di                      ; disable processor interrupts
```
This definition is continued in chunks 14 and 15a.
This code is used in chunk 13a.

We setup the “initial” stack pointer because this will change around once we get into starting up the multiple threads later.

14a ⟨*start hardware init* 13b⟩+≡
```
        ld      sp, #(stack)    ; setup the initial stack pointer
```
This code is used in chunk 13a.

Interrupt mode 1 sends all interrupts through vector `0x0038`, which is what we will use for the IRQ timer.

14b ⟨*start hardware init* 13b⟩+≡
```
        im      1               ; setup interrupt mode 1
```
This code is used in chunk 13a.

For the next bit, we will use a memset function which we define in §15.

Let’s clear the watchdog timer, along with all of the other special hardware. All of the control registers are within the range of `0x5000` through `0x50c0`.

14c ⟨*start hardware init* 13b⟩+≡
```
        ;; clear the special registers
        ld      a, #0x00        ; a  = 0x00
        ld      hl, #(specreg)  ; hl = start of special registers
        ld      b, #(speclen)   ; b  = 0xC0 bytes to zero
        call    memset256       ; 0x5000-0x50C0 will get 0x00
```
This code is used in chunk 13a.

Now clear the sprite registers...

14d ⟨*start hardware init* 13b⟩+≡
```
        ;; clear sprite registers
        ld      a, #0x00        ; a  = 0x00
        ld      hl, #(sprtbase) ; hl = start of sprite registers
        ld      b, #(sprtlen)   ; b  = 0x10 16 bytes
        call    memset256       ; 0x4ff0-0x4fff will get 0x00
```
This code is used in chunk 13a.

Now clear the screen/video ram...

14e ⟨*start hardware init* 13b⟩+≡
```
        ;; clear the screen ram
        call    cls             ; clear the screen RAM
```
This code is used in chunk 13a.

Next, we will need to clear the user ram. This should look very similar, since it needs to do something similar. This is a one-time use thing, so we won't bother making it a callable method. (You will never need to do this once the system is running.)

Similalarly to the above, we need to clear 4 blocks of 256 bytes of ram.

15a ⟨*start hardware init* 13b⟩+≡

```
        ;; clear user ram
        ld      hl, #(ram)      ; hl = base of RAM
        ld      a, #0x03        ; a = 0
        ld      b, #0x02        ; b = 2 blocks of 256 bytes to clear
        call    memsetN         ; clear the blocks
```

This code is used in chunk 13a.

Once we're done with everything, we need to do some pac-specific setup for the interrupt hardware on the machine. Basically we just need to set an interrupt vector and turn on the interrupts externally.

15b ⟨*start enable interrupts* 15b⟩≡

```
        ;; setup pac interrupts
        ld      a, #0xff        ; fill register 'a' with 0xff
        out     (0x00), a       ; send the 0xff to port 0x00
        ld      a, #0x01        ; fill register 'a' with 0x01
```

This definition is continued in chunk 15c.
This code is used in chunk 13a.

Now we just need to enable interrupts, both in the cpu and in the external mechanism.

15c ⟨*start enable interrupts* 15b⟩+≡

```
        ld      (irqen), a      ; enable the external interrupt mechanism.
        ei
```

This code is used in chunk 13a.

Okay... at this point, we're ready to do something real on the machine. Everything has been set up to a state that is now known.

3.2 Display Splash Screen

We just want to display a little something while we wait for things to start up.

(80 bytes code, 67 bytes data)

16 ⟨*start splash screen* 16⟩≡

```
                ; Splash screen!
  .splash:
          call    guicls

          ; draw out the llama!
          ld      hl, #(llama1)   ; top half of llama
          ld      bc, #0x0d09
          ld      a, #(LlamaC)
          call    putstrB
          ld      hl, #(llama2)   ; bottom half of llama
          inc     c
          call    putstrB

          ; draw out the copyright notice and version info
          ld      hl, #(cprt1)
          ld      bc, #0x060f
          ld      a, #0x00        ; black text
          call    putstrB         ; top black border

          ld      bc, #0x0611
          call    putstrB         ; bottom black border

          ld      hl, #(cprt1)
          ld      a, #0x14        ; yellow text
          ld      bc, #0x0610
          call    putstrB         ; 'Alpaca OS...'

          ld      hl, #(cprt2)
          ld      a, #0x0b        ; cyan text
          ld      bc, #0x041e
          call    putstrB         ; '(C) 2003...

          ld      hl, #(cprt3)
          ld      bc, #0x0200
          call    putstrC         ; email addy
```

This code is used in chunk 13a.

17a ⟨*Init splash data* 17a⟩≡
```
llama1:
        .byte   0x02, (LlamaS+0), (LlamaS+1)    ; first row of llama
llama2:
        .byte   0x02, (LlamaS+2), (LlamaS+3)    ; second row of llama
cprt1:
        .byte   0x10
        .ascii  " Alpaca OS v0.8 "
cprt2:
        .byte   0x14
        .ascii  "/2003 Jerry Lawrence"
cprt3:
        .byte   0x18
        .ascii  "alpacaOS@umlautllama.com"
```
This code is used in chunk 102.

3.3 Initialize Tasks

This is covered in /S/refsec:tasksysinit. This just serves as a hook into that section of this document.

17b ⟨*start initialize tasks* 17b⟩≡
⟨*Task System Initialization* 39b⟩
This code is used in chunk 13a.

3.4 Start Runtime

Eventually replace this with the task executor.

18 ⟨*start runtime* 18⟩≡

```
        ;; start runtime

        ; set up sprite 1 as the flying llama
        ld      ix, #(sprtbase)
        ld      a, #(LlamaFS*4)
        ld      0(ix), a
        ld      a, #(3)         ; decent llama color
        ld      1(ix), a

        ;; set up sprite 2 and 3
        ld      ix, #(sprtbase)
        ld      a, #4           ;(hardcoded for now)
        ld      2(ix), a
        ld      4(ix), a
        ld      a, #(3)         ;0x12
        ld      3(ix), a
        ld      5(ix), a

foo:
jp overfoo
        ; fill the screen with a random character
        ld      hl, #vidram
        ld      b, #0x02
        call    rand
        and     #0x0f   ; mask
        add     #0x30   ; base character
        call    memsetN

foo42:
        ; draw a text string
        ld      hl, #(tstr)
        ld      bc, #0x0101
        ld      a, #0x09
        call    putstrB

        ld      bc, #0x1c01
        ld      a, #0x18
        call    textright
        call    putstrA
        call    putstrC

        ld      hl, #(tstr)
        ld      bc, #0x0000
        ld      a, #0x12
        call    textcenter
```

```
        call    putstrA
        call    putstrC

        jp foo

tstr:
        .byte   13
        .ascii  "Hello, world!"
        ; attempt to colorize the background too.

overfoo:

        ; do a lissajous on the screen with the first sprite (arrow cursor)
        ;; X
        ld      ix, #(spritecoords)
        ld      bc, (timer)
        rlc     c       ; *2
        rlc     c       ; *2
        call    sine
        rrca
        and     #0x7f
        add     #0x40
        ld      0(ix), a
        ;; Y
        ld      bc, (timer)
        ;rlc    c
        call    cosine
        rrca
        and     #0x7f
        add     #0x40
        ld      1(ix), a

jp foo
        ; do sprite two now..
        ;; X
        ld      ix, #(spritecoords)
        ld      bc, (timer)
        rlc     c       ; *2
        call    sine
        rrca
        and     #0x7f
        add     #0x40
        ld      2(ix), a
        ;; Y
        ld      bc, (timer)
        rlc     c       ; *2
        call    cosine
        rrca
        and     #0x7f
        add     #0x40
```

```
        ld      3(ix), a

        ; and sprite 3 while we're at it...
        ;; x
        ld      ix, #(spritecoords)
        ld      bc, (timer)
        ld      d, c
        rlc     c
        rlc     c
        call    sine
        rrca
        rrca
        and     #0x3f
        add     a, d
        ld      4(ix), a

        ;; Y
        ld      bc, (timer)
        rlc     c       ; *2
        rlc     c       ; *2
        rlc     c       ; *2
        call    sine
        rrca
        and     #0x7f
        add     #0x40
        ld      5(ix), a

foo2:
        ld      a, (0x4d00)
        add     #6
        ld      b, a
        ld      a, (0x4d01)
        add     #8
        ld      c, a
        call    xy2offsB

        ld      ix, #0x4d00
        ld      a, 2(ix)

        inc     0(ix)           ; x
        bit     4, 0(ix)
        jp      Z, .over

        inc     1(ix)
        ld      0(ix), #0x00
        bit     4, 1(ix)
        jp      Z, .over
        ld      1(ix), #0x00    ; y
        inc     2(ix)           ; color
.over:
```

```
        push    bc
        ld      bc, #colram
        add     hl, bc
        pop     bc
        ld      (hl), a

jp foo

        ; try to hug a screen refresh
        ld      bc, #1
        call    sleep

        jp      foo
        halt
```

Root chunk (not used in this document).

Chapter 4

Kernel Services and API

This chapter describes and defines the interface that tasks use to access the services of the OS kernel.

The services provided by the kernel are provided through the `RST` calls of the Z80 processor. There are 8 of these calls, as well as an interrupt routine that the Z80 provides. The interrupt routine is used by the task switcher, and is described in §8, however an overview of the 8 RST functions is provided next.

Each of these start 8 bytes off from the previous, so we need to be sure that we don't overwrite previous ones, as well as be sure that we start each of them at the right location. We can fill these with five `nops`, but instead, we'll use the `.org` directive on following calls. We just need to be sure that we don't use more than 8 bytes for each of these.

4.1 RST 00H - Startup/Reboot

This is the startup/reboot call. This will setup the system and restart it appropriately according to the initialization routines as defined and implemented in §3. We will just call that routine from here.

The basic initialization starts off at `0x0000` in ROM. This doubles as the implementation for `RST 00`. So we need to be sure that we are at `0x0000`. This simply jumps to the `.startup` routine.

22 ⟨*RST 00 implementation* 22⟩≡

```
.org 0x000
.reset00:                        ; RST 00 - Init
        jp .start
```

This code is used in chunk 102.

4.2 RST 08H - Semaphores

Semaphore control

23a ⟨*RST 08 implementation* 23a⟩≡
```
  .org 0x0008
  .reset08:                           ; RST 08 - Semaphore control
          ret
```
This code is used in chunk 102.

4.3 RST 10H - TBD

TBD

23b ⟨*RST 10 implementation* 23b⟩≡
```
  .org 0x0010
  .reset10:                           ; RST 10 - TBD
          ret
```
This code is used in chunk 102.

4.4 RST 18H - TBD

TBD

23c ⟨*RST 18 implementation* 23c⟩≡
```
  .org 0x0018
  .reset18:                           ; RST 18 - TBD
          ret
```
This code is used in chunk 102.

4.5 RST 20H - TBD

TBD

23d ⟨*RST 20 implementation* 23d⟩≡
```
  .org 0x0020
  .reset20:                           ; RST 20 - TBD
          ret
```
This code is used in chunk 102.

4.6 RST 28H - TBD

TBD

24a ⟨*RST 28 implementation* 24a⟩≡
```
  .org 0x0028
  .reset28:                        ; RST 28 - TBD
          ret
```
This code is used in chunk 102.

4.7 RST 30H - TBD

TBD

24b ⟨*RST 30 implementation* 24b⟩≡
```
  .org 0x0030
  .reset30:                        ; RST 30 - TBD
          ret
```
This code is used in chunk 102.

4.8 RST 38H - VBlank handler

VBLANK IRQ interrupt. This should never be called directly by a task. We will simply jump to the `.isr` function from here, which sits after the below NMI handler, in ROMspace.

24c ⟨*RST 38 implementation* 24c⟩≡
```
  .org 0x0038
  .reset38:                        ; RST 38 - Vblank Interrupt Service Routine
          jp      .isr
```
This code is used in chunk 102.

4.9 NMI handler

We're not using an NMI in this implementation, but we'll leave this here in case we want to use it in the future. This sits at `0x0066`, 38 bytes from the `RST 38` handler. We're basically wasting this space, but we might come back later and fill it in or just drop the NMI handler altogether. Regardless, this handler is here even though it's not used in Pac/Pengo hardware.

24d ⟨*NMI implementation* 24d⟩≡
```
  .org 0x0066
  .nmi:                            ; NMI handler
          retn
```
This code is used in chunk 102.

Chapter 5

Semaphores

This chapter describes how the semaphores are managed in ALPACA.

THESE DON'T SEEM TO WORK PROPERLY YET.

NOTE: We also should disable task switching and/or interrupts when we're locking a semaphore.

5.1 RAM allocation

For now, each semaphore is a single byte. We have 16 allocated for the system, which should be more than enough for four tasks.

These are located at `semabase` in ram.

25 ⟨*Semaphore RAM* 25⟩≡

```
        ; semaphores
    semabase        = (ram + 0x0ce0)
    semamax         = (semabase + 0x0F)
```

This code is used in chunk 102.

5.2 Locking a Semaphore

An attempt to lock a semaphore that is already locked will result in the task blocking until the semaphore is released.

We'll do some rudimentary range limiting on `A` by anding the passed-in semaphore number in the accumulator with `0x0F`, since we only have 16 semaphores.

We then will load `HL` with the base address of the semaphore ram, then add in the above offset onto it.

Once it is released, it will re-set the semaphore, then return to the task.

26 ⟨*Semaphore lock implementation* 26⟩≡

```
      ;; semalock - lock a semaphore
          ;               in      a       which semaphore to lock
          ;               out     -
          ;               mod     -
  semalock:
              ; set aside registers
          push    af
          push    bc
          push    hl
              ; set up the address
          and     #0x0f           ; limit A to 0..15
          ld      c, a            ; c is the current semaphore number
          ld      b, #0x00        ; make sure that b=0   (bc = 0x00SS)
          ld      hl, (semabase)  ; hl = base address
          add     hl, bc          ; hl = address of this semaphore
  .sl2:
          bit     1, (hl)
          jr      NZ, .sl2        ; while it's set, loop
              ; set the bit
          set     1, (hl)         ; lock the semaphore
              ; restore registers
          pop     hl
          pop     bc
          pop     af
              ; return
          ret
```

This code is used in chunk 102.

5.3 Releasing a Semaphore

Releasing a semaphore is even easier than locking one.

Just like the above, we'll do some rudimentary range limiting on `A` by anding the passed-in semaphore number in the accumulator with `0x0F`, since we only have 16 semaphores.

We then will load `HL` with the base address of the semaphore ram, then add in the above offset onto it.

Then we simply clear the bit.

We can eventually combine the two of these if we want, to save a few bytes. Even easier, just after the `res` we can jump to just after the `set` in the above routine... that will save 1 or 2 bytes, but increase obfuscation quite a bit, so we won't do that just yet...

27 ⟨*Semaphore release implementation* 27⟩≡

```
      ;; semarel - release a semaphore
          ;               in      a       which semaphore to release
          ;               out     -
          ;               mod     -
  semarel:
              ; set aside registers
          push    af
          push    bc
          push    hl
              ; set up the address
          and     #0x0F           ; limit A to 0..15
          ld      c, a            ; c is the current semaphore number
          ld      b, #0x00        ; b=0   (bc = 0x000S)
          ld      hl, (semabase)  ; hl = base address
          add     hl, bc          ; hl = address of this semaphore
              ; clear the semaphore
          res     1, (hl)         ; clear the bit
              ; restore registers
          pop     hl
          pop     bc
          pop     af
              ; return
          ret
```

This code is used in chunk 102.

Chapter 6

Message Queue

This chapter describes how all of the messaging in the system is handled.

6.1 Message Format

TBD

6.2 Queue Implementation

Two pointers are maintained into the Message queue; the head and tail pointers. There is also a variable which contains the number of messages currently in the queue. These variables are global for all tasks, and thus the mechanisms for queueing and dequeueing messages into the system are provided by the kernel.

28 ⟨*Message RAM* 28⟩≡

```
        ; messages
    msgbase         = (ram + 0x0ca0)
    msgmax          = (msgbase + 0x003f)
```

This code is used in chunk 102.

6.2.1 Queueing a Message

We need a way to continue adding messages onto the queue while circulating around the ram buffer, so we will have a ram buffer that is 256 bytes large, so that we can just `AND` the offset with `0x00FF` to determine the correct offset into the message queue.

1. If number of messages is greater than 256, fail.
2. Store the message at the RAM location that the tail pointer references
3. Increment the tail pointer
4. `AND` the tail pointer with `0x00FF`
5. Add the tail pointer with the base of the message queue
6. increment the number of messages

6.2.2 Dequeueing a Message

Similarly, we need a way to pop a message off of the queue, so a similar process is used.

1. If number of messages is 0, fail
2. Set the message at the head pointer aside
3. Increment the head pointer
4. `AND` the head pointer with `0x00FF`
5. Add the head pointer with the base of the message queue
6. Decrement the number of messages
7. Return the message

Chapter 7

Memory Management

This chapter describes how all of the memory management (allocation and free) is performed within the system.

7.1 Memory Maintenance Structures

7.2 Memory Acquisition (malloc)

7.3 Memory Release (free)

Chapter 8

Interrupt Service Routine

This chapter describes the Interrupt Sercice Routine within the kernel. This chapter covers the basic Timer as well as the whole task switching routine.

8.1 ISR Overall View

Here is the overall view of the interrupt service routine, which gets called 60 times a second, when the `VBLANK` happens in the video hardware:

31a ⟨*Interrupt Service Routine implementation* 31a⟩≡

```
.isr:
⟨Interrupt disable interrupts and save regs 31b⟩
⟨Interrupt clear the watchdog 32b⟩
⟨Interrupt incrememnt global timer 32d⟩
⟨Interrupt task management 41a⟩
⟨Interrupt enable interrupts and restore regs 32a⟩
```

This code is used in chunk 102.

We need to disable interrupts, both in the CPU was well as in the external interrupt mechanism. In the process of doing this, we will dirty up a few registers, so we might as well save them aside in here also.

31b ⟨*Interrupt disable interrupts and save regs* 31b⟩≡

```
        di                      ; disable interrupts (no re-entry!)
        push    af              ; store aside some registers
        xor     a               ; a = 0
        ld      (irqen), a      ; disable external interrupt mechanism
        push    bc
        push    de
        push    hl
        push    ix
        push    iy
```

This code is used in chunk 31a.

Later on, we'll need to turn interrupts back on, and restore those registers.

32a ⟨*Interrupt enable interrupts and restore regs* 32a⟩≡

```
        ; restore the registers
pop     iy
pop     ix
pop     hl
pop     de
pop     bc
ld      a, #0x01        ; a = 1
ld      (irqen), a      ; enable external interrupt mechanism
pop     af
ei                      ; enable processor interrupts
reti                    ; return from interrupt routine
```

This code is used in chunk 31a.

Anyway, we've still got a `0` loaded into `a` from the above disabling, so we can just send that over to the watchdog as well.

Dealing with the watchdog timer in here prevents the user code (tasks) from having to deal with it at all. The original intention of the watchdog reset hardware is described in §2.1.

32b ⟨*Interrupt clear the watchdog* 32b⟩≡

```
ld      (watchdog), a   ; kick the dog
```

This code is used in chunk 31a.

Also, while in the interrupt routine we want to increment the global timer variable.

The timer is a value in RAM that gets updated by the IRQ/Vblank routine.

32c ⟨*Timer RAM* 32c⟩≡

```
        ; timer counter (word)
timer           = (ram + 21)
```

This code is used in chunk 102.

32d ⟨*Interrupt incrememnt global timer* 32d⟩≡

```
ld      bc, (timer)     ; bc = timer
inc     bc              ; bc++
ld      (timer), bc     ; timer = bc
```

This code is used in chunk 31a.

We could try to do the timer the following way instead, which is fewer bytes of asm, but would only increment the lower byte of the timer, which we don't want. Our current timer is 16 bits, which means that it is only good for about 18 minutes before it overflowed. If we only used 8 bits, our timer would overflow after four seconds. Conversely, a 24 bit timer would last for roughly 77 hours, while a 32 bit timer would last for roughly 821 days... almost three years.

33 ⟨*bad timer* 33⟩≡

```
        ; timer valid for only 4 seconds:
        ld      hl, #(timer)    ; hl = &timer
        inc     (hl)            ; inc the lower 8 bits of the timer.
```

Root chunk (not used in this document).

Future changes to the OS will include an updated timer with a 16 bit "epoch counter" which will give us this 821 day uptime capability, but until then, 18 minutes is probably longer than we'll go before we crash anyway. ;)

And that's the basics. Without the task switching, the above is a useful and fully functional ISR. The sections that follow will add in the task switching.

8.2 Task Switching

The tasks will run in the foreground, just going about their business. These tasks will be interrupted and switched out by the Task Manager from within the Interrupt routine. This will control how much time each task gets, managing their stacks, and all of that fun stuff. Tasks can also give up their remaining time if they are done, waiting for IO or a timer to complete or what have you.

The task switcher is also the backend for the `exec` and `kill` routines, which are described in §10. That is to say that when a task is instantiated with the `exec` command, or a task slot is cleared with the `kill` command, it really only sets flags directly from those commands. All of the work of setting up the task to run in a task slot is handled here in this routine.

The task switcher will also be the backend for the `sleep` routine, once that is implemented correctly.

8.2.1 Design

The design described here supports up to four concurrently running tasks, selected from up to 256 tasks available in the program ROM. There can be multiple instances of the same task running.

Each of the four tasks has its own space in RAM for their own stack and local variables. Each task gets `0x00c0` or (192) bytes of ram which they can use for stack and local variables. Being that the tasks will be written in asm, this should hopefully be more than enough.

There is a variable in RAM, `ramBase` which points to the base of RAM for the currently running task. Tasks will need to define their local variables with reference to this value. Once a task is started, this value will not change.

34 ⟨*Task Constants* 34⟩≡
```
        stacksize       = 192           ; number of bytes per stack
```
This definition is continued in chunk 38c.
This code is used in chunk 102.

And here's where we'll define the stack ram itself:

35a ⟨*Task Stack RAM* 35a⟩≡

```
              ; stack regions for the four tasks
        stackbottom     = (stack-(stacksize*4)) ; 192 bytes (bottom of stack 3)
        stack3          = (stack-(stacksize*3)) ; 192 bytes
        stack2          = (stack-(stacksize*2)) ; 192 bytes
        stack1          = (stack-(stacksize*1)) ; 192 bytes
        stack0          = (stack-(0))           ; top of space - sprite ram
```

This code is used in chunk 102.

This leaves `0x4c00` thru `0x4cff` for program/user ram.

We need to be able to access the above values from the program easily, so we'll set up a table in ROM.

35b ⟨*Task Switch ROM* 35b⟩≡

```
              ; table of stack/user RAM usage (stacks, ram)
  stacklist:
        .word   stack0
        .word   stack1
        .word   stack2
        .word   stack3
        .word   stackbottom
```

This code is used in chunk 102.

The way this table is used is twofold. To find the initial stack pointer for a task slot, just index into the stacklist (`(task slot number) * 2`) bytes in. To find the value to put in `ramBase`, just go to the next item in the array. (`((task slot number + 1) * 2`).

Task Slot Indexes

There are two bytes in RAM per slot that the kernel uses to keep track of the task running in those slots, as well as a way for the task slots to be controlled. These are the `slotIdx` and `slotCtrl` arrays.

The task slot indexes (`slotId`) show which task is loaded in which task slot. This is a single byte (8 bit) index into the `tasklist`, which we will define later.

35c ⟨*Task RAM* 35c⟩≡

```
              ; which task is in which slot (index into tasklist)
        slotIdx         = (ram + 0) ; 4 bytes, one per slot
        slotIdx0        = (ram + 0)
        slotIdx1        = (ram + 1)
        slotIdx2        = (ram + 2)
        slotIdx3        = (ram + 3)
```

This definition is continued in chunks 36–38.
This code is used in chunk 102.

To define these as 'open', we use the following constant:

36a ⟨*Task RAM* 35c⟩+≡
```
        slotOpen        = 0xff
```
This code is used in chunk 102.

Here are the bytes to control each slot. By setting flags in these slots, the ISR will do different things to the slot.

36b ⟨*Task RAM* 35c⟩+≡
```
            ; control information for each slot (to be handled by switcher)
        slotCtrl        = (ram + 4) ; 4 bytes, one per slot
        slot0Ctrl       = (ram + 4)
        slot1Ctrl       = (ram + 5)
        slot2Ctrl       = (ram + 6)
        slot3Ctrl       = (ram + 7)
```
This code is used in chunk 102.

And here are the bits we can set for the control:

First of all, if bit 7 is set, we know that the slot is in use.

36c ⟨*Task RAM* 35c⟩+≡
```
        C_InUse         = 7
```
This code is used in chunk 102.

If bit 4 is set, then the lower four bits are for extenstion commands. This means that if a task wants to perform these actions on the slot, it will set bit 4, and one of the lower three bits.

Bit 0 is the command to kill the task running in that slot. Bit 1 is the command to start up the task in that slot. Bit 2 is the command to relinquish the remaining time for this slot. (Force a task switch, regardless of time left for the slot.)

36d ⟨*Task RAM* 35c⟩+≡
```
        C_EXT0          = 4
            killSlot    = 0
            execSlot    = 1
            sleepSlot   = 2
```
This code is used in chunk 102.

When a task is switched out, we really only need to store the current stack pointer for that slot. That stack pointer is stored somewhere in the `slotSP` array. *NOTE*: the stack pointer location for the currently running slot does not contain valid data. For example, if Slot 2 is active, then `slotSP2` contains invalid data.

37a ⟨*Task RAM* 35c⟩+≡
```
        ; stack pointers for the four slots
        slotSP          = (ram + 8)  ; 8 bytes, two per slot
        slotSP0         = (ram + 8)
        slotSP1         = (ram + 10)
        slotSP2         = (ram + 12)
        slotSP3         = (ram + 14)
```
This code is used in chunk 102.

When a task is running, we need a way to tell it what the base of ram for it is. A task will define its variables in ram with reference to this base pointer. The task can look at `ramBase` to retrieve this data pointer. For example, a task may have one word stored in `(ramBase) + 0`, and a byte stored in `(ramBase) + 2`. This enables tasks to have their own distinct memory blocks so that you can accurately run the same task code multiple times, without them interfering.

37b ⟨*Task RAM* 35c⟩+≡
```
        ; Base of ram for the currently active slot.
        ramBase         = (ram + 16)    ; word
```
This code is used in chunk 102.

We also have one flag which the switcher uses to keep track of the state of the slots. This is the `taskFlag` byte.

37c ⟨*Task RAM* 35c⟩+≡
```
            ; various flags about the task switcher system
        taskFlag        = (ram + 18)    ; byte
```
This code is used in chunk 102.

The lower four bits will show if a slot is in use. If this bit is set, the slot is in use.

37d ⟨*Task RAM* 35c⟩+≡
```
        slot0use        = 0
        slot1use        = 1
        slot2use        = 2
        slot3use        = 3
```
This code is used in chunk 102.

And the fun one. If the `taskActive` flag is set, then the task switching system is running. Clear this, and no switching will take place.

37e ⟨*Task RAM* 35c⟩+≡
```
        taskActive      = 7
```
This code is used in chunk 102.

And of course, the switcher needs to know which slot is the currently active slot. This is contained in the `taskSlot` byte.

38a ⟨*Task RAM* 35c⟩+≡

```
            ; the currently active slot number
    taskSlot        = (ram + 19)    ; byte
```

This code is used in chunk 102.

8.2.2 Task Slot Timing

Each slot will be alotted a certain amount of time. This will change for each slot based on if it is "sleeping", or based on the priority of the task. Or at least, that's how it will be in the future. For now, this will be equally distributed, and requested priorities are ignored. Also, for now, the "sleep" command is dumb, and will just loop within the specified task. Future implementations of "sleep" in the task switching system will interrupt other tasks when the sleep timer expires, to insure that correct timing is given to time-specific tasks.

The switcher will count down the number of ticks that the current slot has before it needs to switch it out. This value is simply set when a task is switched in, and decremented subsequent times through the task switching code. This `slotTime` value can only be up to 255, which is fine, considering that this is about four seconds. Generally, each task should only be run for about 5-10 clock ticks.

38b ⟨*Task RAM* 35c⟩+≡

```
            ; how many ticks does this slot have before it gets swapped out
    slotTime        = (ram + 20)    ; byte
```

This code is used in chunk 102.

For phase one, we will always use a predefined time per task. Make this larger to really show how processing switches from one task to the other. For now, making this around `4` should be plenty. (4/60ths or 1/15th of a second)

38c ⟨*Task Constants* 34⟩+≡

```
    slotTicks       = 4     ; number of ticks per slot to start with
```

This code is used in chunk 102.

8.2.3 Task Search / Task List

Future versions of the OS might include a routine that scans through ROM to find available tasks to run them. Thiw will allow for ROMs, cartridges, or banks to be switched in while the system is live.

In the future, this will produce a `0` terminated list of pointers to the headers in RAM, but for now, we will just have this so-called `tasklist` in ROM.

This is just a list of the headers, terminated with a `0`

39a ⟨*Task List* 39a⟩≡

```
          ; list of all tasks available, null terminated
  tasklist:
          .word t0header
          .word t1header
          .word t2header
          .word t3header
          .word 0x0000
```

This code is used in chunk 102.

8.2.4 Task System Initialization

Now the initialization. This sets it up such that the above ram locations have been initialized properly, and the task switcher in §8.2 knows that the task slot is empty.

First, we need clear the flags, to insure that all of the slots are open, and that the task switcher is disabled.

39b ⟨*Task System Initialization* 39b⟩≡

```
          ;; initialize tasks
              ; clear flags
          xor     a                ; a = 0
          ld      (taskFlag), a    ; clear all task flags
```

This definition is continued in chunks 39 and 40.
This code is used in chunk 17b.

We initialize the stack pointers. This will get replaced in the task switcher, but for now, we will initialize it in here as well. We'll just set them all to 0x0000

39c ⟨*Task System Initialization* 39b⟩+≡

```
              ; clear the dormant stack pointers (set all four to 0x0000)
          xor     a                ; a = 0
          ld      b, #8            ; 8 bytes (4 one-word variables)
          ld      hl, #(slotSP)    ; base of slot stack pointers
          call    memset256        ; clear it
```

This code is used in chunk 17b.

We set all of the task slots as "open" in the slot index pointers as well. We do this by setting the indexes to the special constant, `openslot`, defined above.

40a ⟨*Task System Initialization* 39b⟩+≡
```
        ; set all slots as open
    ld      a, #(slotOpen)  ; a = openslot
    ld      b, #4           ; 4 bytes
    ld      hl, #(slotIdx)  ; base of slot index bytes
    call    memset256
```
This code is used in chunk 17b.

Now we need to clear out all of the control bytes as well.

40b ⟨*Task System Initialization* 39b⟩+≡
```
        ; clear control bytes
    xor     a               ; a = 0
    ld      b, #4           ; 4 bytes
    ld      hl, #(slotCtrl) ; base of slot control bytes
    call    memset256
```
This code is used in chunk 17b.

We also need to set the `taskSlot` variable to something.

40c ⟨*Task System Initialization* 39b⟩+≡
```
        ; clear taskSlot
    xor     a               ; a = 0
    ld      (taskSlot), a   ; taskSlot = 0
```
This code is used in chunk 17b.

Finally, enable the task switcher.

40d ⟨*Task System Initialization* 39b⟩+≡
```
        ; enable the task switcher
    ld      hl, (taskFlag)
    set     #taskActive, (hl)       ; set the flag
```
This code is used in chunk 17b.

8.3 Task Slot Management Mechanism

This section defines the basic overall view of the task slot management routines of the Interrupt Service Routine. The various things that can happen within this framework are defined in §**??** and §**??**.

First, we need the wrapper which checks to see if the task switching is active. We simply check the `taskActive` bit of the `taskFlag` RAM byte. If the flag was zero (`Z`) the bit is not set, and we need to skip over the control flag check routine and the task switching routine. to the `.doneTask` label.

41a ⟨*Interrupt task management* 41a⟩≡

```
                ;; task management stuff
                ; check for disabled switching
        ld      hl, (taskFlag)
        bit     #taskActive, (hl)       ; check to see if task switching is on
        jr      Z, .doneTask            ; jp over if switching is disabled
  ⟨Interrupt check control flags 41b⟩
  ⟨Interrupt attempt to switch to next task 43⟩
  .doneTask:
```

This code is used in chunk 31a.

8.3.1 Control Flag Check

Before we change active task slots, we need to check the control flags for all of the slots to see if they need to be maintained.

41b ⟨*Interrupt check control flags* 41b⟩≡

```
        ; check to see if any of the control flags are set
                ; loop throgh all slots
                ; check for kill
                ; check for sleep
                ; check for start
```

This code is used in chunk 41a.

42 ⟨*notes* 42⟩≡

```
GUI task should always be running (task 0)
never kill the gui task
for now, the gui task is just a tight loop, slot 0

slotMask = 0x03
current slot (taskSlot) is always valid
taskSlot = 0x4c??

**go to next valid slot:

**Start new task:
    move SP into (slotSP)[curr]
    set SP to base of slot
    push (start point of task)
    push (extra registers as 0x00)
    move SP into (slotSP)[thisslot]
    set this slot as 'in use'
    clear slot flags
    move (slotSP)[curr] into SP

**Kill,start, relinquish
    all require a flags check loop before the main loop
    (every time in the ISR, check the flags for all slots)

    (tmp) = 0
  .loop
    check ctrl reg for changes:
        if set to kill:
            mark slot as not in use
        if set to start:
            **start new task
    inc (tmp)
    if (tmp) < 4, jp .loop

    if set to relinquish time:
    set (slottime) to 1
```

Root chunk (not used in this document).

8.3.2 Task Switch Routine

First, we need to wrap the task switcher with a check to see if it is time[1] to switch task slots yet. We simply look at the `slotTime` byte to see if it is greater than `0`. If it is greater than zero, then we skip over the task switching routine.

If we are still greater than zero, we skip over the task switch. Then we just reload `C` with the slot time, decrement it, and store it back in Ram.

We could save a few bytes, and decrement the counter before we do anything, but that would mean that the above sleep would set the time left to `1` instead of `0` which seems wrong. For the few extra bytes that it saves us, it's more intuitive to do it this way.

43 ⟨*Interrupt attempt to switch to next task* 43⟩≡

```
            ;; check to see if we need to task switch yet
        ld      hl, #slotTime           ; hl = time address
        ld      c, (hl)                 ; c = current time for active slot
            ; check the current value
        xor     a                       ; a = 0
        cp      c                       ; is C >=0?  ( Carry set )
        jp      C, .noSwitch            ; still greater than zero?
⟨Interrupt switch to next task 44⟩
.noSwitch:
            ; decrement the slot timer
        ld      hl, #slotTime           ; hl = time address
        ld      c, (hl)                 ; c = current time for active slot
        dec     c                       ; current time --
        ld      (hl), c                 ; store the current time
```

This code is used in chunk 41a.

[1]...wait for it...

XXX Need to break this up and document it better XXX

44 ⟨*Interrupt switch to next task* 44⟩≡

```
            ;; change to next dormant task (or this one...)
  .tsNext:
        ld      a, (taskSlot)           ; a = current task slot (a is try)
        ld      e, a                    ; de = current slot
  .tsloop1:
        inc     a                       ; ++try
        and     a, #slotMask            ; try &= 0x03
        ld      hl, #(slotCtrl)         ; hl = slotCtrl base
        ld      c, a
        ld      b, #0x00                ; bc = task number
        add     hl, bc                  ; hl = control for this task
        bit     #C_InUse, (hl)          ; check the flag
        jr      NZ, .tsloop1            ; if not active, inc again
            ; compare selected task with "current"
        ld      a, e                    ; A = current (again)
        cp      c                       ; compare A(curr) and C(try)
        jr      Z, .overslot1           ; skip this next bit if we're there
  .storeTheSP:
            ; snag the SP into IX
        ld      ix, #0x0000             ; zero ix
        add     ix, sp                  ; ix = SP

            ; setup HL as ram location to store SP
        ld      hl, #(slotSP)           ; hl = base of slotSP array
        ld      d, #0x00                ; de = current slot
        rlc     e                       ;    = current slot * 2
                                        ; bc still contains the try value
        add     hl, de                  ; hl = base of current slot SP
        push    ix                      ; de
        pop     de                      ;    = SP
            ; store the current SP
        ld      (hl), e                 ; (hl) =
        inc     hl
        ld      (hl), d                 ;      = de   (really SP)
  .loadInTheSP:
            ; swap in the new SP
        ld      d, #0
        ld      e, c                    ; de = new slot number
        rlc     e                       ;    = new slot number * 2
        ld      hl, #(slotSP)           ; hl = base of slotSP array
        add     hl, de                  ; hl = base of new slot SP
            ; snag it and shove it into place
        ld      e, (hl)                 ; de =
        inc     hl
        ld      d, (hl)                 ;    = new sp
        ld      h, d                    ; hl =
        ld      l, e                    ;    = sp
```

```
        ld      sp, hl                  ; new SP!
.setupVars:
            ; set up reference variables
        ld      a, c                    ; a = c
        ld      (taskSlot), a           ; taskSlot = new slot number
            ; set up ramBase
        ld      hl, #(stackList)        ; hl = base of stackList array
        ld      e, c                    ; e = new slot
        inc     e                       ; e = new slot + 1
        rlc     e                       ; e = (new slot + 1) * 2
        ld      d, #0                   ; de = (new slot + 1) * 2
        add     hl, de                  ;    = index of this slot + 1 word
        ld      c, (hl)                 ; bc =
        inc     hl
        ld      b, (hl)                 ;    = new ramBase item
        ld      hl, #(ramBase)
        ld      (hl), c                 ; ramBase =
        inc     hl
        ld      (hl), b                 ;         = correct value!
.overslot1:
        ld      hl, #slotTime           ; hl = time address
        ld      (hl), #slotTicks        ; reset the ticks for this task
```

This code is used in chunk 43.

Chapter 9

The Core Task

This chapter describes the core task. This is the task that deals with doing all of the things that the ISR doesn't have time to do, or doesn't need to do as often. For example, checking I/O.

This task will eventually be replaced with the GUI task. This task occupies task slot 0. This leaves 3 task slots to be used by user code.

9.1 Core Runtime Loop

This loop will be run by the OS, and will eventually contain things like timer and message distribution, as well as joystick movement-to-position as well as IO-to-click message handlers.

46 ⟨*.coretask implementation* 46⟩≡

```
.coretask:
        ; set up sprite 1 as the flying llama
        ld      ix, #(sprtbase)
        ld      a, #(LlamaFS*sprtMult)
        ld      sprtIndex(ix), a
        ld      a, #(3)                 ; decent llama color
        ld      sprtColor(ix), a

        ;; set up sprite 2 and 3
        ld      ix, #(sprtbase)
        ld      a, #4           ;(hardcoded for now)
        ld      2+sprtIndex(ix), a
        ld      4+sprtIndex(ix), a
        ld      a, #(3)         ;0x12
        ld      2+sprtColor(ix), a
        ld      4+sprtColor(ix), a

foo:
```

```
; do a lissajous on the screen with the first sprite (arrow cursor)
;; X
ld      ix, #(spritecoords)
ld      bc, (timer)
rlc     c       ; *2
rlc     c       ; *2
call    sine
rrca
and     #0x7f
add     #0x40
ld      sprtIndex(ix), a
;; Y
ld      bc, (timer)
;rlc    c
call    cosine
rrca
and     #0x7f
add     #0x40
ld      sprtColor(ix), a

; try to hug a screen refresh
ld      bc, #1
call    sleep

jp      foo
halt
```

This code is used in chunk 102.

Chapter 10

Task Exec

This chapter describes how a task is started up within the ALPACA system. We also describe how a task needs to be formated within the ROMspace such that the kernel can find the tasks, run them and interact with them.

10.1 Task Format Header

This is basically just a simple header that has all of the information that the OS needs to work with a task. The four byte cookie is there for the task searcher, which is not currently implemented, but will be in future versions of ALPACA.

- 4 bytes - magic cookie `0xc9 0x4a 0x73 0x4c` ('ret' 'J' 's' 'L') (for the searcher)
- 1 byte - task format version `0x01` (version 1)
- 1 byte - requested priority. This is the number of timeslices the task wants at a particular run between switching out.
- 2 bytes - pointer to an pascal/asciz string for task name. The data this points to should consist of a byte with the string length in it, followed immediately by that string, null terminated.
- 2 bytes - task entry point. This is just the address to the task's main routine.

10.2 Task Entry Point

This is the routine that the "exec" will jump to when the task is started up. This routine should not return. It should end with a `halt` opcode, and possibly call the `kill` routine to dequeue itself from the system, and open the slot.

10.3 Start Task (exectask)

This will take in two values. First is a value which specifies which task to run. This is used as an index into the `tasklist` array, defined in §8.2.3. Secondly, it takes in a value which specifies in which slot to run that task.

The name "execute" is really a misnomer. The task will not really be executed in this section, but rather, the task will be scheduled to be run in a specified task slot. This task will then be started within the task switcher routine, in §8.2.

And this is why all of the information about actually starting a task or killing a task (later on) is covered in §8.

In a nutshell, to start up a task in a slot, we set the task number into `A`, and the slot into `D`. This will set the control register for the specific slot at `taskctrl[d]` with the task to run. We just need to be sure that bit 7 of the task number is clear. We also need to limit the slot to [0..3].

49 ⟨*Exec start implementation* 49⟩≡

```
      ;; execstart - starts up a new task
          ;               in      E       task number to start
          ;               in      D       task slot to use (0..3)
          ;               out     -
          ;               mod     -
  execstart:
              ; save registers we're using
          push    af
          push    de
          push    bc
          push    hl
              ; limit E (task) to 127
          res     7, e            ; limit task number to 127
              ; limit D (slot)
          ld      a, d            ; a=d
          and     #0x03           ; slot is 0,1,2, or 3
          ld      c, a            ; c=a
          ld      b, #0x00        ; b=0x00, bc = 0x000S
              ; set the control value
          ld      hl, #(taskctrl) ; set up the control register
          add     hl, bc          ; hl = base + offset
          ld      (hl), e         ; taskctrl[d] = e
              ; restore the registers
          pop     hl
          pop     bc
          pop     de
          pop     af
              ; return
          ret
```

This code is used in chunk 102.

10.4 Stop Task (kill)

We also might need a way to stop or “kill” a task. In traditional *NIX systems, “kill” sends a signal to the program to tell it to stop running. We don’t have signals (yet), so we will just implement this in the same mindset as the above. We will just signal the task switcher to remove the references to this task. Again, this does not happen in here, but rather, over in §8.2.

We basically just set the value in the appropriate

50 ⟨*Exec kill implementation* 50⟩≡

```
      ;; execkill - kills a running task
          ;               in      D       task slot to kill
          ;               out     -
          ;               mod     -
  execkill:
              ; save registers we're using
          push    af
          push    de
          push    bc
          push    hl
              ; limit D (slot) and shove it into C
          ld      a, d            ; a=d
          and     #0x03           ; slot is 0,1,2, or 3
          ld      c, a            ; c=a
          ld      b, #0x00        ; b=0x00,  bc = 0x000S
              ; set the control value
          ld      hl, #(taskctrl) ; set up the control register
          add     hl, bc          ; hl = base + offset
          ld      (hl), #(killslot)       ; taskctrl[d] = KILL!
              ; restore the registers
          pop     hl
          pop     bc
          pop     de
          pop     af
              ; return
          ret
```

This code is used in chunk 102.

10.5 Sleep for some time (sleep)

One thing that is very useful to have is a way for a process to wait for a specified amount of time. This is accomplished through this “sleep” command. The task puts the number of ticks to wait (60 per second) into `BC` then calls this routine.

Future versions might relinquish remaining clock cycles to other tasks by this communicating somehow to the task switcher, but this one just sits in a loop, waiting for the clock to be the right value.

But for this version, we will compute the timeout `current time + ticks to wait`, and just store it in `BC` while we loop.

The loop simply loads the current time into `HL`, then subtracts `BC` from it. We then compare it with a `sbc`, and loop if we’re not there yet.

NOTE that this is not completely accurate. There might be 1-N more ticks between when this routine returns past when you expect it to return. This is due to the multitasking nature of /OS. Your timer might be up, but another task has the processing cycles currently. As soon as we have the cpu again, we will time out and return.

51 ⟨*Exec sleep implementation* 51⟩≡

```
        ;; sleep - wait a specified number of ticks
        ;               in      bc      number of ticks to wait
        ;               out     -
        ;               mod     -
sleep:
            ; set side some registers
        push    bc
        push    af
        push    hl
            ;; this is where we would set the flag for
            ;; the exec system to relinquish the rest of our time.
            ; compute the timeout into BC
        ld      hl, (timer)     ; hl = timer
        add     hl, bc          ; hl += ticks to wait
        push    hl              ; bc =
        pop     bc              ;    = hl
.slp:
            ; loop until the timeout comes
        ld      hl, (timer)     ; hl = current time
        sbc     hl, bc          ; set flags
        jp      M, .slp         ; if (HL >= BC) then JP .slp2
            ; restore the registers
        pop     hl
        pop     af
        pop     bc
            ; return
        ret
```

This code is used in chunk 102.

Here's what I had originally wrote. Notice that it keeps the timeout persistant by keeping it on the stack. This required an extra pop and push for each iteration through the loop, and also required an extra push and pop wrapped around that.

The above implementation only uses the stack to move the value of `hl` over into `bc`, and that happens once per call.

52 ⟨*original sleep implementation* 52⟩≡

```
        ;; oldsleep - wait a specified number of ticks
        ;               in      bc      number of ticks to wait
        ;               out     -
        ;               mod     -
oldsleep:
            ; set aside some registers
        push    bc
        push    af
        push    hl
            ; compute the timeout into HL
        ld      hl, (timer)     ; hl = timer
        add     hl, bc          ; hl += ticks to wait
        push    hl              ; top of stack now contains the timeout value
.slp2:
            ; loop until the timeout comes
        pop     hl              ; restore hl...
        push    hl              ;           ...and shove it back on the stack
        ld      bc, (timer)     ; bc = current time
        sbc     hl, bc          ; set flags
        jr      P, .slp2        ; if (HL < BC) then JR .slp2
        pop     hl
            ; restore the registers
        pop     hl
        pop     af
        pop     bc
            ; return
        ret
```

Root chunk (not used in this document).

Chapter 11

Task 0: Pac Tiny User Interface (PTUI)

This chapter implements the GUI for the system called "PTUI". This task will be loaded into the system as task number `0`.

11.1 Graphics

As you can see in figures 11.1 - 11.4, The GUI widgets, window ornamentations, and cursor are stored in various locations in the graphics banks. (Use the checkerboard image to identify the sprite numbers for each of the graphical elements.

The tile graphics in bank 1, figure 11.1 are pretty basic. It simply contains alphanumerics for text, as well as the widgets needed for the windows.

The sprite graphics in bank 2, figure 11.3 contain just the cursor that the joystick will be moving around for the GUI.

These banks are the same for Pac-Man and Pengo. Pengo has one other character bank, and one other sprite bank, both of which are not used for this task.

Figure 11.1: Graphics Bank 1: Tile Graphics

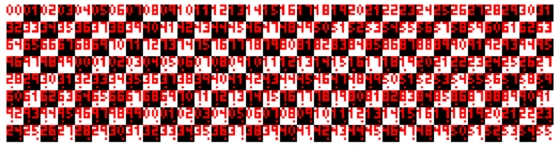

Figure 11.2: Bank 1 Checkerboard Image

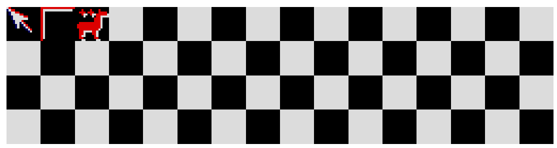

Figure 11.3: Graphics Bank 2: Sprite Graphics

This next set of blocks defines those graphical element reference numbers, as well as the colors for those elements.

54 ⟨*Task 0 constants* 54⟩≡

```
          ; GUI constants
  ⟨GUI cursor and wallpaper 55a⟩
  ⟨GUI flags 55b⟩
  ⟨GUI frame and dragbar 56⟩
  ⟨GUI widgets 57⟩
  ⟨GUI widget types 58a⟩
```

This code is used in chunk 58b.

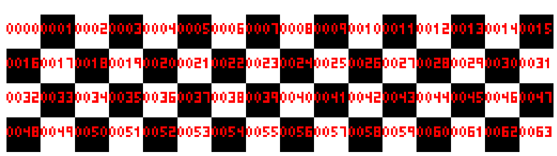

Figure 11.4: Bank 2 Checkerboard Image

11.1.1 Cursor and Wallpaper

55a ⟨*GUI cursor and wallpaper* 55a⟩≡

```
; cursor and wallpaper
    PcursorS        =    0  ; sprite 0 for the cursor
    PcursorC        =    9  ; color 9 for the cursor

    CrosshFS        =    1  ; crosshair for window movement
    CrosshC         = 0x09  ; crosshair color

    PwpS            =  162  ; wallpaper sprite
    PwpC            = 0x10  ; wallpaper color 0x13- blues

    LlamaC          = 0x10  ; llama color  (might be the same as PwpC above)
    LlamaS          = 0x7b  ; base of llama tile
    LlamaFS         =    2  ; llama floating sprite
    CprtC           = 0x14  ; copyright color 11
```

This code is used in chunk 54.

11.1.2 Flags

55b ⟨*GUI flags* 55b⟩≡

```
; flags
    F_Noframe       =    1  ; no frame in render (hard flag)
    F_Frame         =    2  ; frame in render (hard flag)

    F_Dirty         =    1  ; frame needs redraw (soft flag)
    F_Focus         =    2  ; frame is capturing focus currently
```

This code is used in chunk 54.

11.1.3 Frame and Dragbar

56 ⟨*GUI frame and dragbar* 56⟩≡

```
; -- frame widgets --
    ; close
    PcloseS        =  128  ; close widget sprite
    PcloseCS       =    1  ; close widget selected color (5)
    PcloseCU       = 0x1e  ; close widget unselected color

    ; raise
    PraiseS        =  131  ; raise widget sprite
    PraiseCS       =    1  ; raise widget selected color (5)
    PraiseCU       =  0xc  ; raise widget unselected color

; -- frame ornaments --
    PfrmTSel       =    9  ; dragbar text selected color 0x14 0xb
    PfrmTUns       =    1  ; dragbar text unselected color

    PfrmCSel       =    1  ; frame selected color
    PfrmCUns       = 0x1e  ; frame unselected color

    ; bottom corners
    PSWcornS       =  138  ; southwest corner
    PSEcornS       =  139  ; southeast corner

    ; top corners
    PNWcornS       =    1  ; northwest corner 140
    PNEcornS       =    1  ; northeast corner 141

    ; top bar
    PfN_W          =  129  ; top left      (145 or 129)
    PfN_N          =   32  ; top center    (146 or 32)
    PfN_E          =  130  ; top right     (147 or 130)

    ; left bar
    PfW_N          =  132  ; left top
    PfW_W          =  133  ; left center
    PfW_S          =  134  ; left bottom

    ; right bar
    PfE_N          =  135  ; right top
    PfE_E          =  136  ; right center
    PfE_S          =  137  ; right bottom

    ; bottom bar
    PfS_W          =  142  ; bottom left
    PfS_S          =  143  ; bottom center
    PfS_E          =  144  ; bottom right
```

This code is used in chunk 54.

11.1.4 Widgets

57 ⟨*GUI widgets* 57⟩≡

```
; widgets
    PwC             =    1  ; generic widget color
    PwBGS           =  127  ; window background sprite

    ; button
    PwbLuS          =  148  ; [    button left unselected sprite
    PwbRuS          =  149  ;    ] button right unselected sprite

    ; selected button
    PwbLsS          =  150  ; [[   button left selected sprite
    PwbRsS          =  151  ;   ]] button right selected sprite

    ; checkbox
    PwcuS           =  152  ; [ ] checkbox unselected sprite
    PwcsS           =  153  ; [X] checkbox selected sprite

    ; radio box
    PwruS           =  154  ; ( ) radio unselected sprite
    PwrsS           =  155  ; (X) radio selected sprite

    ; slider
    PwsnS           =  156  ; === slider notch sprite
    PwsbS           =  157  ; =|= slider bar sprite

    ; progress bar
    PwpoS           =  158  ;     progress bar open sprite
    PwpfS           =  159  ; ### progress bar filled sprite

    ; spin
    PwHsS           =  160  ; <>  horizontal spin controller
    PwVsS           =  161  ; ^v  vertical spin controller
```

This code is used in chunk 54.

11.1.5 Widget Type Flags

58a ⟨*GUI widget types* 58a⟩≡

```
          ; Widget Types (for the frame-widget table)

          W_End          =    0  ; end of the widget list
          W_Frame        =    1  ; window frame (needs to be first)

           ; frame flags:
           FF_Border     =    1  ; use a border on the frame
           FF_NClose     =    2  ; no close button
           FF_NRaise     =    4  ; no raise button

          W_MButton      =    2  ; momentary button
          W_SButton      =    3  ; sticky button

          W_Radio        =    4  ; radio button (flags is the group number)
          W_Check        =    5  ; check button

          W_SText        =    6  ; static text  (text is the idx of a string)
          W_DText        =    7  ; dynamic text (data is idx of ram)

          W_DInt         =    8  ; dynamic integer (data is idx in the ram)

          W_HSlider      =    9  ; horizontal slider
          W_VSlider      =   10  ; vertical slider

          W_HSpin        =   11  ; horizontal spin
          W_VSpin        =   12  ; vertical spin
```

This code is used in chunk 54.

11.2 Implementation

58b ⟨*Task 0 implementation* 58b⟩≡

```
          ;; Task 0 - PTUI
  ; constants
  ⟨Task 0 constants 54⟩

  ; header
  ⟨Task 0 header 59a⟩

  ; routines
  ⟨Task 0 process routine 59b⟩
```

This code is used in chunk 102.

11.3 Header

59a ⟨*Task 0 header* 59a⟩≡

```
t0header:
        .byte   0xc9, 0x4a, 0x73, 0x4c  ; cookie
        .byte   0x01                    ; version
        .byte   0x04                    ; requested timeslices
        .word   t0name                  ; name
        .word   t0process               ; process function

t0name:
        .byte   6                       ; strlen
        .asciz  "Task 0"                ; name
```

This code is used in chunk 58b.

11.4 Process routine

59b ⟨*Task 0 process routine* 59b⟩≡

```
t0process:
        ld      hl, #(colram)   ; base of color ram
        ld      a, #0x01        ; clear the screen to 0x00
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN         ; do it.

t0p2:
        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x41        ; 'A'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x42        ; 'B'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x43        ; 'C'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        jp      t0p2
        halt
```

This code is used in chunk 58b.

Chapter 12

Task 1: TBD Example

This chapter implements a simple task which will be loaded into the system as task number 1.

60a ⟨*Task 1 implementation* 60a⟩≡
```
        ;; Task 1 - TBD
; header
```
⟨*Task 1 header* 60b⟩
```
; routines
```
⟨*Task 1 process routine* 61⟩

This code is used in chunk 102.

12.1 Header

60b ⟨*Task 1 header* 60b⟩≡
```
t1header:
        .byte   0xc9, 0x4a, 0x73, 0x4c  ; cookie
        .byte   0x01                    ; version
        .byte   0x04                    ; requested timeslices
        .word   t1name                  ; name
        .word   t1process               ; process function

t1name:
        .byte   6                       ; strlen
        .asciz  "Task 1"                ; name
```
This code is used in chunk 60a.

12.2 Process routine

61 ⟨*Task 1 process routine* 61⟩≡

```
t1process:
        ld      hl, #(colram)   ; base of color ram
        ld      a, #0x01        ; clear the screen to blue
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(colram)   ; base of color ram
        ld      a, #0x09        ; clear the screen to red
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        jp      t1process
        halt
```

This code is used in chunk 60a.

Chapter 13

Task 2: TBD Example

This chapter implements a simple task which will be loaded into the system as task number 2.

62a ⟨*Task 2 implementation* 62a⟩≡

```
        ;; Task 2 - TBD
; header
⟨Task 2 header 62b⟩

; routines
⟨Task 2 process routine 63⟩
```

This code is used in chunk 102.

13.1 Header

62b ⟨*Task 2 header* 62b⟩≡

```
t2header:
        .byte   0xc9, 0x4a, 0x73, 0x4c  ; cookie
        .byte   0x01                    ; version
        .byte   0x04                    ; requested timeslices
        .word   t2name                  ; name
        .word   t2process               ; process function

t2name:
        .byte   6                       ; strlen
        .asciz  "Task 2"                ; name
```

This code is used in chunk 62a.

13.2 Process routine

63 ⟨*Task 2 process routine* 63⟩≡

```
t2process:
        ld      hl, #(colram)   ; base of color ram
        ld      a, #0x01        ; clear the screen to 0x00
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x61        ; 'a'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x62        ; 'b'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x63        ; 'c'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        jp      t2process
        halt
```

This code is used in chunk 62a.

Chapter 14

Task 3: TBD Example

This chapter implements a simple task which will be loaded into the system as task number 3.

64a ⟨*Task 3 implementation* 64a⟩≡

```
        ;; Task 3 - TBD
; header
⟨Task 3 header 64b⟩

; routines
⟨Task 3 process routine 65⟩
```

This code is used in chunk 102.

14.1 Header

64b ⟨*Task 3 header* 64b⟩≡

```
t3header:
        .byte   0xc9, 0x4a, 0x73, 0x4c  ; cookie
        .byte   0x01                    ; version
        .byte   0x04                    ; requested timeslices
        .word   t3name                  ; name
        .word   t3process               ; process function

t3name:
        .byte   6                       ; strlen
        .asciz  "Task 3"                ; name
```

This code is used in chunk 64a.

14.2 Process routine

65 ⟨*Task 3 process routine* 65⟩≡

```
t3process:
        ld      hl, #(colram)   ; base of color ram
        ld      a, #0x01        ; clear the screen to 0x00
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x78        ; 'X'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x79        ; 'Y'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x7a        ; 'Z'
        ld      b, #0x04        ; 256*4 = 1k
        call    memsetN

        jp      t3process
        halt
```

This code is used in chunk 64a.

Chapter 15

Utility Functions

This chapter describes and implements a few functions that are usable by tasks, and have some sort of utility value.

15.1 memset256 - set up to 256 bytes of memory to a certian byte

Here we will implement a function that sets a region of memory to a certian value. Load the value into `a`, the base address into `hl`, and the number of bytes into `b`. We might want to use this in task space, so we'll make it a utility function.

66 ⟨*Utils memset256 implementation* 66⟩≡

```
        ;; memset256 - set up to 256 bytes of ram to a certain value
        ;               in      a       value to poke
        ;               in      b       number of bytes to set 0x00 for 256
        ;               in      hl      base address of the memory location
        ;               out     -
        ;               mod     hl, bc
memset256:
        ld      (hl), a         ; *hl = 0
        inc     hl              ; hl++
        djnz    memset256       ; decrement b, jump to memset256 if b>0
        ret                     ; return
```

This code is used in chunk 102.

15.2 memsetN - set N blocks of memory to a certian byte

Here we will implement a function that sets a region of memory to a certian value. Load the value into `a`, the base address into `hl`, and the number of blocks of 256 bytes into `b`. We might want to use this in task space, so we'll make it a utility function.

67 ⟨*Utils memsetN implementation* 67⟩≡

```
        ;; memsetN - set N blocks of ram to a certain value
        ;               in      a       value to poke
        ;               in      b       number of blocks to set
        ;               in      hl      base address of the memory location
        ;               out     -
        ;               mod     hl, bc
memsetN:
        push    bc              ; set aside bc
        ld      b, #0x00        ; b = 256
        call    memset256       ; set 256 bytes
        pop     bc              ; restore the outer bc
        djnz    memsetN         ; if we're not done, set another chunk.
        ret                     ; otherwise return
```

This code is used in chunk 102.

15.3 cls - clear the screen

The screen ram is two chunks of ram from `0x4000` through `0x43FF` as well as `0x4400` through `0x47FF`. We will clear these to black.

We'll basically nest two loops, both using the `djnz`. The inner loop happens in the memset function. The outer loop happens 8 times, since we need to do 256 bytes 8 times. (`djnz` only looks at 8 bits of register 'b'.)

68 ⟨*Utils cls implementation* 68⟩≡

```
        ;; cls - clear the screen (color and video ram)
        ;               in      -
        ;               out     -
        ;               mod     -
cls:
        push    hl              ; set aside some registers
        push    af
        push    bc

        ld      hl, #(vidram)   ; base of video ram
        ld      a, #0x00        ; clear the screen to 0x00
        ld      b, #0x08        ; need to set 256 bytes 8 times.

        call    memsetN         ; do it.

        pop     bc              ; restore the registers
        pop     af
        pop     hl
        ret                     ; return
```

This code is used in chunk 102.

15.4 guicls - clear the screen to GUI background

Basically, this will just do a `cls`, but it will draw the textured background to the screen insteas of just leaving it blank. The tiles to use for this are defined in the `task0` definition, in §11.1.1.

Due to the fact that we're going to be using a different value for the tile and color, we need to have distinct, seperate loops for the color ram and video ram, unfortunately.

69a ⟨*Utils guicls implementation* 69a⟩≡
```
        ;; guicls - clear the screen to the GUI background
        ;               in      -
        ;               out     -
        ;               mod     -
guicls:
        push    hl              ; set aside some registers
        push    af
        push    bc

        ; fill the screen with the background color
        ld      hl, #(colram)   ; color ram
        ld      a, #(PwpC)      ; color
        ld      b, #0x04        ; 4 blocks
        call    memsetN

        ; fill the screen with the background tile
        ld      hl, #(vidram)   ; character ram
        ld      a, #(PwpS)      ; background tile
        ld      b, #0x04        ; 4 blocks
        call    memsetN

        pop     bc              ; restore the registers
        pop     af
        pop     hl
        ret                     ; return
```
This code is used in chunk 102.

15.5 rand - get a random number

This function returns a pseudorandom number in register A.

We need a byte for persistance, to get the previous Random number we gave out:

69b ⟨*Rand RAM* 69b⟩≡
```
            ; random assistance register (byte)
        randval         = (ram + 23)
```
This code is used in chunk 102.

The algorithm I'm doing here is just a standard mutilating calculation like so:

70a ⟨*calculation* 70a⟩≡

```
new random number = current timer + sine( last random number ) + R
```

Root chunk (not used in this document).

It's just something simple that we can replace with something better later. In the meantime, it should give something reasonably random, although not decently distributed throughout `[0..256]`.

We also will include the memory refresh register, since that one is constantly changing. If our application used sound, and we're on Pac hardware, we could also add in the accumulator registers from the sound hardware as well.

We can pull out the items between `.r01` and `.r02` if we've determined that the `R` register adds nothing useful to the randomization of the system

70b ⟨*Utils rand implementation* 70b⟩≡

```
        ;; rand - get a random number
        ;               in      -
        ;               out     a       random number 0..256
        ;               mod     flags
rand:
            ; set aside registers
        push    hl
        push    bc
            ; compute a random number
        ld      hl, (randval)   ; hl = last random number
        push    hl
        pop     bc              ; bc = hl
        call    sine            ; a = sine (c)
        ld      c, a            ; c = sine ( last value )
.r01:
        ld      a, r            ; a = R
        add     a, c            ; a += sine( last value )
        ld      c, a            ; c = sine( last value ) + R
.r02:
        add     hl, bc          ; rnd += sin ( last value ) + R
        ld      bc, (timer)
        add     hl, bc          ; rnd += timer
        ld      (randval), hl   ; hl = computed random (rnd)
        ld      a, (randval)    ; a = rnd
            ; restore registers
        pop     bc
        pop     hl
            ; return
        ret
```

This code is used in chunk 102.

15.6 sine - return the sine

This function returns the modified sine of the angle passed in in register `C`. It returns this value in register `A`.

To simplify this, instead of expecting rotational angle on a range of `[0..360]` degrees, we will instead expect the rotational angle to be on a range of 256 units per complete circle. We will also return a value from `[-127..127]` instead of `[-1..1]` since we can't work with decimal values easily. This should be good enough for most uses.

71 ⟨*Utils sine implementation* 71⟩≡

```
        ;; sine - get the sine of a
        ;               in      c       value to look up
        ;               out     a       sine value 0..256
        ;               mod     -
sine:
            ; set aside registers
        push    hl
        push    bc
            ; look up the value in the sine table
        ld      hl, #(.sinetab) ; hl = sinetable base
        ld      b, #0x00        ; b = 0
        add     hl, bc          ; hl += bc
        ld      a, (hl)         ; a = sine(c)
            ; restore registers
        pop     bc
        pop     hl
            ; return
        ret
```

This code is used in chunk 102.

Since we're here, we might as well throw in a cosine function as well. We just add `0x7f` onto the angle passed in via `C`, and look up that value in the sine table using the above method.

72 ⟨*Utils cosine implementation* 72⟩≡
```
        ;; cosine - get the cosine of a
        ;               in      c       value to look up
        ;               out     a       cosine value 0..256
        ;               mod     f
cosine:
            ; set aside registers
        push    bc
            ; add 180 degrees, call sine
        ld      a, #0x3f
        add     a, c
        ld      c, a
        call    sine
            ; restore registers
        pop     bc
            ; return
        ret
```
This code is used in chunk 102.

73 ⟨*Utils sine table* 73⟩≡

```
.sinetab:
        .byte   0x80, 0x83, 0x86, 0x89, 0x8c, 0x8f, 0x92, 0x95
        .byte   0x99, 0x9c, 0x9f, 0xa2, 0xa5, 0xa8, 0xab, 0xae
        .byte   0xb1, 0xb4, 0xb6, 0xb9, 0xbc, 0xbf, 0xc2, 0xc4
        .byte   0xc7, 0xc9, 0xcc, 0xcf, 0xd1, 0xd3, 0xd6, 0xd8
        .byte   0xda, 0xdc, 0xdf, 0xe1, 0xe3, 0xe5, 0xe7, 0xe8
        .byte   0xea, 0xec, 0xee, 0xef, 0xf1, 0xf2, 0xf3, 0xf5
        .byte   0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd
        .byte   0xfd, 0xfe, 0xfe, 0xff, 0xff, 0xff, 0xff, 0xff
        .byte   0xff, 0xff, 0xff, 0xff, 0xff, 0xfe, 0xfe, 0xfd
        .byte   0xfd, 0xfc, 0xfb, 0xfb, 0xfa, 0xf9, 0xf8, 0xf7
        .byte   0xf5, 0xf4, 0xf3, 0xf1, 0xf0, 0xee, 0xed, 0xeb
        .byte   0xe9, 0xe8, 0xe6, 0xe4, 0xe2, 0xe0, 0xde, 0xdb
        .byte   0xd9, 0xd7, 0xd5, 0xd2, 0xd0, 0xcd, 0xcb, 0xc8
        .byte   0xc6, 0xc3, 0xc0, 0xbd, 0xbb, 0xb8, 0xb5, 0xb2
        .byte   0xaf, 0xac, 0xa9, 0xa6, 0xa3, 0xa0, 0x9d, 0x9a
        .byte   0x97, 0x94, 0x91, 0x8e, 0x8b, 0x87, 0x84, 0x81
        .byte   0x7e, 0x7b, 0x78, 0x74, 0x71, 0x6e, 0x6b, 0x68
        .byte   0x65, 0x62, 0x5f, 0x5c, 0x59, 0x56, 0x53, 0x50
        .byte   0x4d, 0x4a, 0x47, 0x44, 0x42, 0x3f, 0x3c, 0x39
        .byte   0x37, 0x34, 0x32, 0x2f, 0x2d, 0x2a, 0x28, 0x26
        .byte   0x24, 0x21, 0x1f, 0x1d, 0x1b, 0x19, 0x17, 0x16
        .byte   0x14, 0x12, 0x11, 0x0f, 0x0e, 0x0c, 0x0b, 0x0a
        .byte   0x08, 0x07, 0x06, 0x05, 0x04, 0x04, 0x03, 0x02
        .byte   0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00
        .byte   0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x01, 0x02
        .byte   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09
        .byte   0x0a, 0x0c, 0x0d, 0x0e, 0x10, 0x11, 0x13, 0x15
        .byte   0x17, 0x18, 0x1a, 0x1c, 0x1e, 0x20, 0x23, 0x25
        .byte   0x27, 0x29, 0x2c, 0x2e, 0x30, 0x33, 0x36, 0x38
        .byte   0x3b, 0x3d, 0x40, 0x43, 0x46, 0x49, 0x4b, 0x4e
        .byte   0x51, 0x54, 0x57, 0x5a, 0x5d, 0x60, 0x63, 0x66
        .byte   0x6a, 0x6d, 0x70, 0x73, 0x76, 0x79, 0x7c, 0x7f
```

This code is used in chunk 102.

That table was generated with this perl snippet:

74 ⟨*sinegen.pl* 74⟩≡

```
$across = 8;                # number to print horizontally
$current = $across +1;

print ".sinetab:";
for ( $x=0 ; $x < 256 ; $x++ )
{
    $rads = ($x/255.0) * 6.283185307;
    #printf "%3d %f\n",$x, 128 + 128 *(sin $rads);

    $value = 128 + 128 *(sin $rads);

    if ($current >= $across)
    {
       print "\n\t.word\t";
       $current = 0;
    }
    $current ++;

    printf "0x%02x", $value;
    if ( ($x < 255) && ($current < $across))
    {
       printf ", ";
    }
}
print "\n";
```

Root chunk (not used in this document).

15.7 textcenter - centers text to be drawn

This function modifies the coordinates in `BC` based on the pascal string contained in HL. It simply replaces the value in `B` with a value that will result in the text being centered on the screen.

75 ⟨*Utils textcenter implementation* 75⟩≡

```
        ;; textcenter - adjust the x ordinate
        ;               in      hl      pascal string
        ;               in      b       x ordinate
        ;               in      c       y ordinate  BC ->  0xXXYY
        ;               out     -
        ;               mod     b       adjusted for center
        hscrwide = 14
textcenter:
            ; set aside registers
        push    af
            ; halve the width
        ld      b, (hl)         ; b = length of text
        jp      NC, .tcrr       ; make sure carry is cleared
        ccf
.tcrr:
        rr      b               ; b = half of text length
            ; add on the center position
        ld      a, #hscrwide    ; a = screenwidth/2
        sub     b               ; a = screenwidth/2 - textlength/2
        ld      b, a            ; b = that result
            ; restore registers
        pop     af
            ; return
        ret
```

This code is used in chunk 102.

15.8 textright - right justifies text to drawn

This function modifies the coordinates in `BC` based on the pascal string contained in HL. It simply replaces the value in `B` with a value that will result in the text being right justified off of that location.

76 ⟨*Utils textright implementation* 76⟩≡

```
        ;; textright - adjust the x ordinate
        ;               in      hl      pascal string
        ;               in      b       x ordinate
        ;               in      c       y ordinate  BC ->  0xXXYY
        ;               out     -
        ;               mod     b       adjusted for right
textright:
            ; set aside registers
        push    af
            ; halve the width
        ld      a, b            ; a = start location
        ld      b, (hl)         ; b = length of text
        sub     b               ; a = start loc - length
        ld      b, a            ; b = new position
            ; restore registers
        pop     af
            ; return
        ret
```

This code is used in chunk 102.

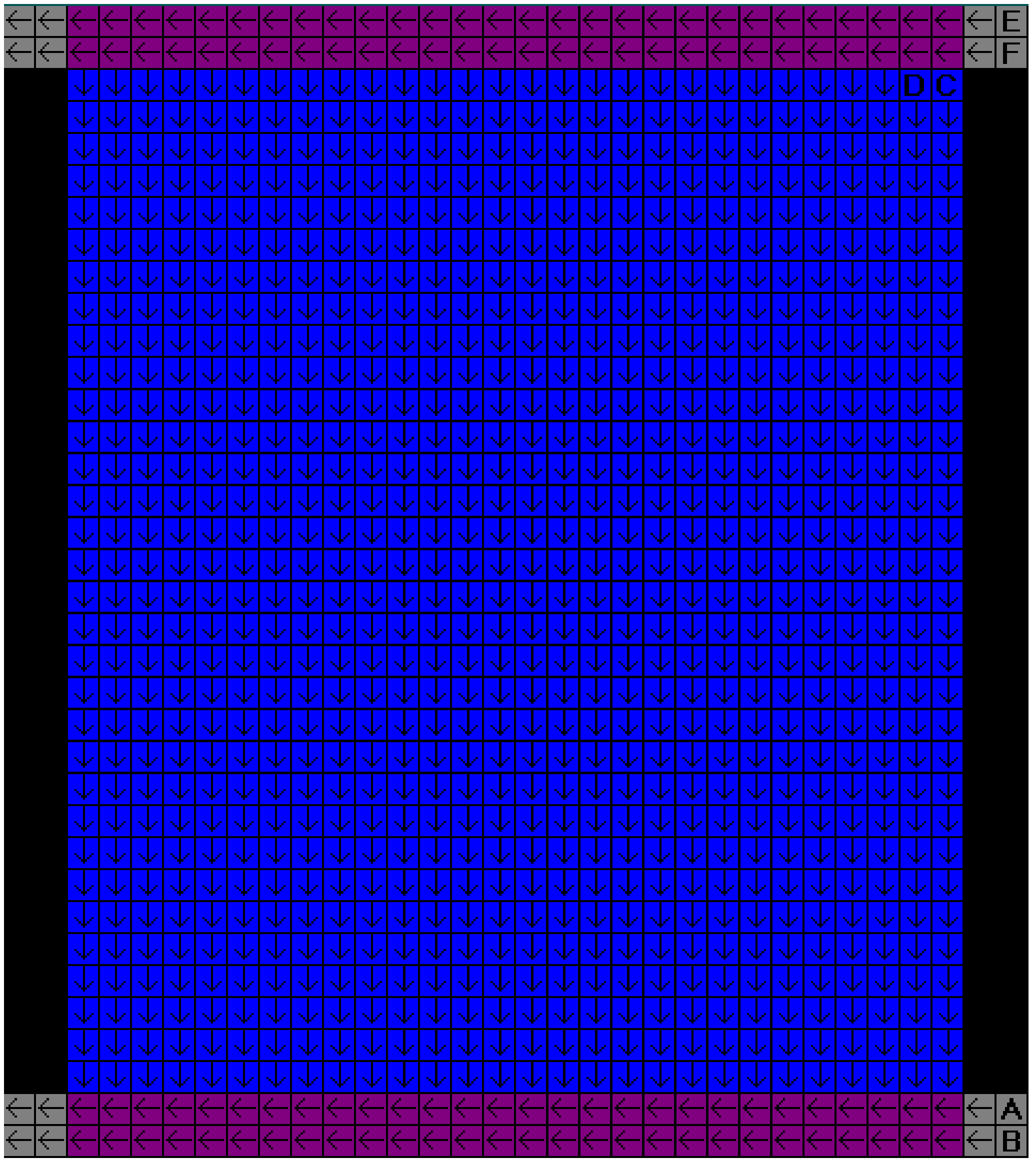

Figure 15.1: Video Screen Layout

15.9 Screen Region A tools

Screen region A is the topmost two rows of characters of the screen. The characters are addressed right-to-left for the top row, then right-to-left for the second row. These are shown in figure 15.1 as the topmost two purple rows "E" and "F".

We now provide routines for converting XY for this region into offsets into the color or video ram, as well as routines for drawing out text.

15.9.1 xy2offsAC - convert X,Y into offsets in screen region A and C

Since regions A and C are pretty muc hthe same thing, we will use the same function for both regions. We will define the bottom two rows ("A" and "B" in figure 15.1) as rows `2` and `3`, while the top two rows, "E" and "F" will be defined as rows `0` and `1`.

78 ⟨*Utils acoffs table* 78⟩≡

```
.acoffs:
        .word   0x03dd          ; Region A row 'E' -> AC row 0
        .word   0x03fd          ; Region A row 'F' -> AC row 1
        .word   0x001d          ; Region C row 'A' -> AC row 2
        .word   0x003d          ; Region C row 'B' -> AC row 3
```

This code is used in chunk 102.

To make the decoding a little easier, we first will define this table of four offset addresses. To decode the offset from the XY position passed in via BC, we use C as the index into this table, then we just add on B to that, and return the computed value in HL.

79 ⟨*Utils xy2offsAC implementation* 79⟩≡

```
        ;; xy2offAC - get the vid/color buffer offset of the X Y coordinates
        ;               in      b       x ordinate
        ;               in      c       y ordinate  BC ->  0xXXYY
        ;               out     hl      offset
        ;               mod     -
xy2offsAC:
            ; set aside registers
        push    bc
        push    de
        push    ix
            ; generate the X component into DE
        ld      d, #0x00        ; d = 0
        ld      e, b            ; e = X
            ; get the base offset
        ld      ix, #(.acoffs)  ; ix = offset table base
            ; add in the y component. (BC)
        ld      b, #0x00        ; zero B (top of BC)
        rlc     c               ; y *= 2
        add     ix, bc          ; offset += index
            ; retrieve that value into HL
        ld      b, 1(ix)
        ld      c, 0(ix)
        push    bc
        pop     hl              ; hl = acroffs[x]
            ; subtract out the X component.
        sbc     hl, de          ; hl -= DE   hl = acoffs[y]-x
            ; restore registers
        pop     ix
        pop     de
        pop     bc
            ; return
        ret
```

This code is used in chunk 102.

15.9.2 putstrA - draw a string on region A of the screen

Since regions A and C are pretty much the same thing, just with different start positions, we will have hooks in here for C to jump into.

80 ⟨*Utils putstrA implementation* 80⟩≡

```
        ;; putstrA - get the vid/color buffer offset of the X Y coordinates
        ;               in      hl      pointer to the string (asciz)
        ;               in      b       x position
        ;               in      c       y position
        ;               in      a       color
        ;               out     -
        ;               mod     -
putstrA:
            ; set aside registers
        push    bc
.psChook:                       ; this is where putstrC joins in...
        push    hl
        push    de
        push    ix
        push    iy
            ; compute the offsets
        push    hl              ; set aside the string pointer
        call    xy2offsAC
        push    hl
        pop     ix              ; move the offset into ix (char ram)
        push    hl
        pop     iy              ; move the offset into iy (color ram)
        ld      de, #(vidram)   ; base of video ram
        add     ix, de          ; set IX to appropriate location in vid ram
        ld      de, #(colram)   ; base of color ram
        add     iy, de          ; set IY to appropriate location in color ram
            ; prep for the loop
        pop     hl
        ld      b, (hl)         ; b is the number of bytes (pascal string)
        inc     hl              ; HL points to the text now
.pstra1:
            ; loop for each character
        ld      c, (hl)         ; c = character
        ld      (ix), c         ; vidram[b+offs] = character
        ld      (iy), a         ; colram[b+offs] = color
            ; adjust pointers
        inc     hl              ; inc string location
        dec     ix              ; dec char ram pointer
        dec     iy              ; dec color ram pointer
        djnz    .pstra1         ; dec b, jump back if not done
            ; restore registers
        pop     iy
        pop     ix
        pop     de
```

```
        pop     hl
        pop     bc
            ; return
        ret
```
This code is used in chunk 102.

15.10 Screen Region C tools

Since region C is addressed similarly to region A, we will discuss that next instead of going into region B. In fact, this section leverages heavily on the previous section.

Screen region C is the bottommost two rows of characters of the screen. The characters are addressed right-to-left for the second-to-bottom row, then right-to-left for the bottom row. These are shown in figure 15.1 as the bottommost two purple rows "A" and "B".

We now provide routines for drawing out text.

15.10.1 putstrC - draw a string on region C of the screen

Since regions A and C are pretty much the same thing, just with different start positions, we simply massage the input position data, and jump into the above `putstrA` function.

81 ⟨*Utils putstrC implementation* 81⟩≡
```
        ;; putstrC - get the vid/color buffer offset of the X Y coordinates
        ;               in      hl      pointer to the string (asciz)
        ;               in      b       x position
        ;               in      c       y position
        ;               in      a       color
        ;               out     -
        ;               mod     -
putstrC:
            ; set aside registers
        push    bc
        inc     c               ; just change indexing 0,1 into 2,3
        inc     c
        jp      .psChook        ; jump back into putstrA
```
This code is used in chunk 102.

15.11 Screen Region B tools

Screen Region B is the main body of the screen. It's characters are addressed from top-to-bottom for the rightmost column, then top-to-bottom for the column just to the left of that, and so on for 28 columns. These are shown in figure 15.1 as the center blue area, starting at column "C", then "D".

We now provide routines for converting XY for this region into offsets into the color or video ram, as well as routines for drawing out text.

15.11.1 xy2offsB - convert X,Y into offsets in screen region B

Since a lot of what we're doing involves interacting with the screen, we might as well have a method in here for converting X,Y (from the upper left) to screen offsets. The offset generated by this can be added to either the base video or color ram to determine screen locations in RAM.

Basically, you load `B` with the X component, and `C` with the Y component. You then call this utility, and the correct offset gets loaded into `HL`. You can then add in the base for video or color ram to draw your characters to the screen, or retrieve information from the screen.

It should be noted that the location `X,Y == (0,0)` is in the upper left of the screen, two character tiles from the top of the visible area of the screen, due to the existence of Region A.

82 ⟨*Utils xy2offsB implementation* 82⟩≡

```
        ;; xy2offsB - get the vid/color buffer offset of the X Y coordinates
        ;               in      b       x ordinate
        ;               in      c       y ordinate  BC ->  0xXXYY
        ;               out     hl      offset
        ;               mod     -
xy2offsB:
            ; set aside registers
        push    af
        push    bc
        push    de
        push    ix
            ; set aside Y for later in DE
        ld      d, #0x00        ; d = 0
        ld      e, c            ; shove Y into E
            ; get the base offset
        ld      ix, #(.scroffs) ; ix = offset table base
            ; add in X component
                ;; XXXXJJJJJ This can probably be shortened if we
                ;;              drop the range check.
        ld      a, b            ; shove X into A
        and     a, #0x1f        ; make sure X is reasonable
```

```
        rlc     a               ; x *= 2
        ld      c, a            ; c = offset * 2
        ld      b, #0x00        ; b = 0
        add     ix, bc          ; ix += bc
            ; retrieve that value into HL
        ld      b, 1(ix)
        ld      c, 0(ix)
        push    bc
        pop     hl              ; hl = scroffs[x]
            ; add in Y component
        add     hl, de          ; hl += DE   hl = scroffs[x]+y
            ; restore registers
        pop     ix
        pop     de
        pop     bc
        pop     af
            ; return
        ret
```

This code is used in chunk 102.

This looks into the following table of screen offsets, which define where each column (left-to-right) starts in the color or video buffers. These just need to be added on to either of those buffer base addresses, then simply add in the y position.

83 ⟨*Utils scroffs table* 83⟩≡

```
  .scroffs:
        .word   0x03a0, 0x0380, 0x0360, 0x0340
        .word   0x0320, 0x0300, 0x02e0, 0x02c0
        .word   0x02a0, 0x0280, 0x0260, 0x0240
        .word   0x0220, 0x0200, 0x01e0, 0x01c0
        .word   0x01a0, 0x0180, 0x0160, 0x0140
        .word   0x0120, 0x0100, 0x00e0, 0x00c0
        .word   0x00a0, 0x0080, 0x0060, 0x0040
```

This code is used in chunk 102.

That table was generated with this perl snippet:

84 ⟨*scroffs.pl* 84⟩≡

```
#!/usr/bin/perl

$wide = 28;
$tall = 36;

# screen offset = .scroffs[x] + y;

$across = 4;
$current = $across +1;

printf ".scroffs:";

for ($x=0 ; $x<$wide ; $x++)
{
    if( $current >= $across)
    {
        print"\n\t.byte\t";
        $current = 0;
    }
    $current++;

    printf "0x%04x", (928 - ($tall-4) * $x);

    if( ($x < $wide) && ($current < $across))
    {
        printf ", ";
    }
}
printf "\n";
```

Root chunk (not used in this document).

15.11.2 putstrB - draw a string on region B of the screen

This is just a simple routine to draw out a pascal string to the screen within the vertical scanning region. (ie not the top two or bottom two rows of the screen, which are addressed differently.

Simply load the color into `A`, the X,Y position into `B`,`C`, and the pointer to the pascal string into `HL`.

In a single loop, it draws out the character and sets the color for the text it is drawing.

It should be noted that there are no safeguards around this, so if your text is longer than 28 characters wide, it will get truncated, and might overwrite program RAM, which is a very bad thing to do.

The code simply sets up the char and color pointers into `IX` and `IY`, and increments them by `-32` for each iteration through the loop, while at the same time, it draws the correct character and color through those pointers.

85 ⟨*Utils putstrB implementation* 85⟩≡

```
        ;; putstrB - get the vid/color buffer offset of the X Y coordinates
        ;               in      hl      pointer to the string (asciz)
        ;               in      b       x position
        ;               in      c       y position
        ;               in      a       color
        ;               out     -
        ;               mod     -
        offsadd = -32
putstrB:
            ; set aside registers
        push    hl
        push    bc
        push    de
        push    ix
        push    iy
        push    hl
            ; compute the offsets
        call    xy2offsB        ; hl = core offset
        push    hl
        pop     ix              ; move the offset into ix (char ram)
        push    hl
        pop     iy              ; move the offset into iy (color ram)
        ld      de, #(vidram)   ; base of video ram
        add     ix, de          ; set IX to appropriate location in vid ram
        ld      de, #(colram)   ; base of color ram
        add     iy, de          ; set IY to appropriate location in color ram
            ; prep for the loop
        pop     hl
        ld      b, (hl)         ; b is the number of bytes (pascal string)
        inc     hl              ; HL points to the text now
        ld      de, #offsadd    ; set up the column offset
```

```
.pstrb1:
            ; loop for each character
        ld      c, (hl)         ; c = character
        ld      (ix), c         ; vidram[b+offs] = character
        ld      (iy), a         ; colram[b+offs] = color
            ; adjust pointers
        inc     hl              ; inc string location
        add     ix, de          ; add in offset into char ram
        add     iy, de          ; add in offset into color ram
        djnz    .pstrb1         ; dec b, jump back if not done
            ; restore registers
        pop     iy
        pop     ix
        pop     de
        pop     bc
        pop     hl
            ; return
        ret
```

This code is used in chunk 102.

Here's an older implementation, which did more stack pushing and popping. It is 54 bytes long, and uses two loops to draw the text. One to draw the text, and one to draw the color.

The previous routine is 47 bytes long, and does it all within one loop.

87 ⟨*Utils 54 byte putstr implementation* 87⟩≡

```
        ;; putstr - get the vid/color buffer offset of the X Y coordinates
        ;               in      iy      pointer to the string (asciz)
        ;               in      b       x position
        ;               in      c       y position
        ;               in      d       color
        ;               out     -
        ;               mod     -
        offsadd = -32
putstr:
            ; set aside registers
        push    hl
        push    af
        push    bc
        push    iy
        push    de
            ; retrieve the offset
        call    xy2offsB        ; hl = core offset
        push    hl              ; store it on the stack
        pop     hl
        push    hl
        ld      de, #(vidram)   ; base of video ram
        add     hl, de          ; set HL to appropriate location in vid ram

            ; draw out the string
        ld      de, #offsadd    ; setup the column offset
        ld      b, (iy)         ; b is the number of bytes (pascal string)
.pstr1:
        inc     iy              ; iy is now the string offset
        ld      a, (iy)         ; a contains a character to draw
        ld      (hl), a         ; send it to the screen
        add     hl, de          ; add in the offset to the screen
        djnz    .pstr1          ; dec b, jump back if not done

            ; set the color
        pop     hl              ; restore offset value
        ld      de, #(colram)   ; base of color ram
        add     hl, de          ; set HL to appropriate location in color ram
        pop     de              ; restore the color info
        ld      a, d
            ; draw up the color
        pop     iy              ; restore the string pointer (for length)
        ld      b, (iy)         ; b is the number of bytes (pascal string)
        ld      de, #offsadd    ; setup the column offset
.pstr2:
```

```
ld      (hl), a         ; fill in the color
add     hl, de          ; add in the offset to the screen
djnz    .pstr2          ; dec b, jump back if not done

    ; restore registers
pop     bc
pop     hl
pop     af
    ; return
ret
```

Root chunk (not used in this document).

15.11.3 mult8 - 8 bit multiply

88 ⟨*mult8 protocode* 88⟩≡

```
        HL=H*E

        LD    L, 0
        LD    D, L    ; L = 0  and  D = 0
        LD    B, 8
MULT:   ADD   HL, HL
        JR    NC, NOADD
        ADD   HL, DE

NOADD:  DJNZ  MULT
```

Root chunk (not used in this document).

Chapter 16

System Errors

This chapter describes how system errors are handled in ALPACA.

The System error routines are formatted similarly to the task routines. When the kernel finds an error during its interrupt routine, it will push the correct address for the error routine then return from the interrupt handler.

Each error routine should disable interrupts, clear the watchdog timer, and draw some kind of informative information on the screen for the user to see.

Errors are currently unimplemented.

Chapter 17

Appendix

Appendix A

Development Schedule

The development cycles for ALPACA have been broken down into a few phases. Each of the phases will be completed before then next one will be started.

A.1 Phase 1

- task startup with hardcoded entry points
- task switching with hardcoded priorites/delays
- init and process routines for tasks

A.2 Phase 2

- task exec with ROM Task searcher
- simple message queue (not useful)

A.3 Phase 3

- task switching with wait(0), requested priorities
- more advanced message queue
- shutdown routine for tasks
- perhaps allow for multiple execs of the same process (this collides with the searcher's functionality)

Appendix B

Hardware memory constants

This chapter lists off all of the addresses for all of the bits of hardware that we will have to deal with. This chapter includes information about Pac-Man as well as Pengo hardware.

B.1 Pac-Man Configuration

92a ⟨*PAC Global Constants* 92a⟩≡

```
        stack           = 0x4ff0
```

This definition is continued in chunks 92–96.
This code is used in chunk 100a.

92b ⟨*PAC Global Constants* 92a⟩+≡

```
        vidram          = 0x4000
        colram          = 0x4400
        ram             = 0x4c00
        dsw0            = 0x5080
        in1             = 0x5040
        in0             = 0x5000
        specreg         = 0x5000
        speclen         = 0x00C0
        sprtbase        = 0x4ff0
        sprtlen         = 0x0010
```

This code is used in chunk 100a.

The bits for player 1 joystick

93a ⟨*PAC Global Constants* 92a⟩+≡
```
p1_port         = in0
p1_up           = 0
p1_left         = 1
p1_right        = 2
p1_down         = 3
```
This code is used in chunk 100a.

The bits for player 2 joystick

93b ⟨*PAC Global Constants* 92a⟩+≡
```
p2_port         = in1
p2_up           = 0
p2_left         = 1
p2_right        = 2
p2_down         = 3
```
This code is used in chunk 100a.

The bits for joystick buttons. Since Pac hardware has no fire buttons, we'll just absorb the start buttons instead.

93c ⟨*PENGO Global Constants* 93c⟩≡
```
p1_bport        = in1
p1_b1           = 5
p2_bport        = in1
p1_b1           = 6
```
This definition is continued in chunks 96–99.
This code is used in chunk 100b.

The bits for start buttons

93d ⟨*PAC Global Constants* 92a⟩+≡
```
start_port      = in1
start1          = 5
start2          = 6
```
This code is used in chunk 100a.

The bits for coin inputs

93e ⟨*PAC Global Constants* 92a⟩+≡
```
coin_port       = in0
coin1           = 5
coin2           = 6
coin3           = 7
```
This code is used in chunk 100a.

And the bits for cabinet, test and service switches:

94a ⟨*PAC Global Constants* 92a⟩+≡
```
        rack_port       = in0
        racktest        = 4
        svc_port        = in1
        service         = 4
        cab_port        = in1
        cabinet         = 7
```
This code is used in chunk 100a.

B.1.1 Sprite Hardware

This constants 8 pairs of two bytes:

- byte 1, bit 0 - Y flip
- byte 1, bit 1 - X flip
- byte 1, bits 2-7 - sprite image number
- byte 2 - color

When drawing the sprite, we need to multiply the sprite number to clear the XY flip bits.

94b ⟨*PAC Global Constants* 92a⟩+≡
```
        sprtMult        = 4
```
This code is used in chunk 100a.

And we should have offset numbers, to help out with IX and IY indexing of the sprite array.

94c ⟨*PAC Global Constants* 92a⟩+≡
```
        sprtColor       = 1
        sprtIndex       = 0
```
This code is used in chunk 100a.

`sprtXFlip` defines the byte offset which contains the X flip bit. `bitXFlip` defines the bit number to use if using `SET` or `RES` opcodes. `valXFlip` defines the value to use if creating a byte to poke in.

94d ⟨*PAC Global Constants* 92a⟩+≡
```
        sprtXFlip       = 0
        bitXFlip        = 0
        valXFlip        = 1
        sprtYFlip       = 0
        bitYFlip        = 1
        valYFlip        = 2
```
This code is used in chunk 100a.

Here's the base of the sprite RAM.

95a ⟨*PAC Global Constants* 92a⟩+≡
```
          spritebase      = 0x4ff0
```
This code is used in chunk 100a.

And there are 8 sprites total:

95b ⟨*PAC Global Constants* 92a⟩+≡
```
          nsprites        = 0x08
```
This code is used in chunk 100a.

And for the coordinates, these are xy pairs for 8 sprites.

95c ⟨*PAC Global Constants* 92a⟩+≡
```
          spritecoords    = 0x5060
```
This code is used in chunk 100a.

B.1.2 Sound Hardware

Three voices. Voice 1:

95d ⟨*PAC Global Constants* 92a⟩+≡
```
          v1_acc          = 0x5040
          v1_wave         = 0x5045
          v1_freq         = 0x5050
          v1_vol          = 0x5055
```
This code is used in chunk 100a.

Voice 2:

95e ⟨*PAC Global Constants* 92a⟩+≡
```
          v2_acc          = 0x5046
          v2_wave         = 0x504a
          v2_freq         = 0x5056
          v2_vol          = 0x505a
```
This code is used in chunk 100a.

Voice 3:

95f ⟨*PAC Global Constants* 92a⟩+≡
```
          v3_acc          = 0x504b
          v3_wave         = 0x504f
          v3_freq         = 0x505b
          v3_vol          = 0x505f
```
This code is used in chunk 100a.

B.1.3 Enablers

96a ⟨*PAC Global Constants* 92a⟩+≡
```
        irqen           = 0x5000
        sounden         = 0x5001
        flipscreen      = 0x5003
        coincount       = 0x5007
        watchdog        = 0x50C0
```
This code is used in chunk 100a.

B.1.4 Extras for Pac

96b ⟨*Pac Global Constants* 96b⟩≡
```
        strtlmp1        = 0x5004
        strtlmp2        = 0x5005
        coinlock        = 0x5006
```
Root chunk (not used in this document).

B.2 Pengo Configuration

96c ⟨*PENGO Global Constants* 93c⟩+≡
```
        stack           = 0x8ff0
```
This code is used in chunk 100b.

96d ⟨*PENGO Global Constants* 93c⟩+≡
```
        vidram          = 0x8000
        colram          = 0x8400
        ram             = 0x8800
        dsw0            = 0x9040
        in1             = 0x9080
        in0             = 0x90c0
        specreg         = 0x9000
        speclen         = 0x00ff
        sprtbase        = 0x8ff2
        sprtlen         = 0x0010
```
This code is used in chunk 100b.

The bits for player 1 joystick

96e ⟨*PENGO Global Constants* 93c⟩+≡
```
        p1_port         = in0
        p1_up           = 0
        p1_down         = 1
        p1_left         = 2
        p1_right        = 3
```
This code is used in chunk 100b.

The bits for player 2 joystick

97a ⟨*PENGO Global Constants* 93c⟩+≡

```
p2_port        = in1
p2_up          = 0
p2_down        = 1
p2_left        = 2
p2_right       = 3
```

This code is used in chunk 100b.

The bits for joystick buttons

97b ⟨*PENGO Global Constants* 93c⟩+≡

```
p1_bport       = in0
p1_b1          = 7
p2_bport       = in1
p1_b1          = 7
```

This code is used in chunk 100b.

The bits for start buttons

97c ⟨*PENGO Global Constants* 93c⟩+≡

```
start_port     = in1
start1         = 5
start2         = 6
```

This code is used in chunk 100b.

The bits for coin inputs

97d ⟨*PENGO Global Constants* 93c⟩+≡

```
coin_port      = in0
coin1          = 4
coin2          = 5
coin3          = 6
```

This code is used in chunk 100b.

And the bits for service

97e ⟨*PENGO Global Constants* 93c⟩+≡

```
svc_port       = in1
service        = 4
```

This code is used in chunk 100b.

B.2.1 Sprite Hardware

This constants 8 pairs of two bytes:

- byte 1, bit 0 - Y flip
- byte 1, bit 1 - X flip
- byte 1, bits 2-7 - sprite image number
- byte 2 - color

When drawing the sprite, we need to multiply the sprite number to clear the XY flip bits.

98a ⟨*PENGO Global Constants* 93c⟩+≡
```
          sprtMult        = 4
```
This code is used in chunk 100b.

And we should have offset numbers, to help out with IX and IY indexing of the sprite array.

98b ⟨*PENGO Global Constants* 93c⟩+≡
```
          sprtColor       = 1
          sprtIndex       = 0
```
This code is used in chunk 100b.

`sprtXFlip` defines the byte offset which contains the X flip bit. `bitXFlip` defines the bit number to use if using `SET` or `RES` opcodes. `valXFlip` defines the value to use if creating a byte to poke in.

98c ⟨*PENGO Global Constants* 93c⟩+≡
```
          sprtXFlip       = 0
          bitXFlip        = 0
          valXFlip        = 1
          sprtYFlip       = 0
          bitYFlip        = 1
          valYFlip        = 2
```
This code is used in chunk 100b.

Here's the base of the sprite RAM.

98d ⟨*PENGO Global Constants* 93c⟩+≡
```
          spritebase      = 0x8ff2
```
This code is used in chunk 100b.

And there are 8 sprites total:

98e ⟨*PENGO Global Constants* 93c⟩+≡
```
          nsprites        = 0x06
```
This code is used in chunk 100b.

And for the coordinates, these are xy pairs for 8 sprites.

99a ⟨*PENGO Global Constants* 93c⟩+≡
```
        spritecoords    = 0x9022
```
This code is used in chunk 100b.

B.2.2 Sound Hardware

Three voices. Voice 1:

99b ⟨*PENGO Global Constants* 93c⟩+≡
```
        v1_wave         = 0x9005
        v1_freq         = 0x9011
        v1_vol          = 0x9015
```
This code is used in chunk 100b.

Voice 2:

99c ⟨*PENGO Global Constants* 93c⟩+≡
```
        v2_wave         = 0x900a
        v2_freq         = 0x9016
        v2_vol          = 0x901a
```
This code is used in chunk 100b.

Voice 3:

99d ⟨*PENGO Global Constants* 93c⟩+≡
```
        v3_wave         = 0x900f
        v3_freq         = 0x901b
        v3_vol          = 0x901f
```
This code is used in chunk 100b.

B.2.3 Enablers

99e ⟨*PENGO Global Constants* 93c⟩+≡
```
        irqen           = 0x9040
        sounden         = 0x9041
        flipscreen      = 0x9043
        coincount       = 0x9044
        watchdog        = 0x9070
```
This code is used in chunk 100b.

B.2.4 Extras for Pengo

99f ⟨*PENGO Global Constants* 93c⟩+≡
```
        palbank         = 0x9042
        collutbank      = 0x9046
        spritebank      = 0x9047
```
This code is used in chunk 100b.

Appendix C

The .asm File

This is where we gather together all of the asm blocks defined above into two cohesive .asm files.

C.1 Pac-Man ASM

100a ⟨*pacalpaca.asm* 100a⟩≡

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PacAlpaca.asm
;
;  ALPACA: A Multitasking operating system for Pac-Man Z80 arcade hardware
;
⟨commontop.asm 101⟩
⟨PAC Global Constants 92a⟩
⟨commonbottom.asm 102⟩
```

Root chunk (not used in this document).

C.2 Pengo ASM

100b ⟨*pengoalpaca.asm* 100b⟩≡

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PengoAlpaca.asm
;
;  ALPACA: A Multitasking operating system for Pengo Z80 arcade hardware
;
⟨commontop.asm 101⟩
⟨PENGO Global Constants 93c⟩
⟨commonbottom.asm 102⟩
```

Root chunk (not used in this document).

C.3 Common Top

101 ⟨*commontop.asm* 101⟩≡

```
;  Written by
;       Scott "Jerry" Lawrence
;       alpaca@umlautllama.com
;
;  This source file is covered by the LGPL:
;
```

⟨*license short version* 125⟩

```
;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;           This file is machine generated.  Do not edit it by hand!
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


        .title alpaca
        .module alpaca

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; some constants:
```

This code is used in chunk 100.

C.4 Common Bottom

102 ⟨*commonbottom.asm* 102⟩≡

```
; constants for the task system
⟨Task Constants 34⟩


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RAM allocation:
⟨Task RAM 35c⟩
⟨Timer RAM 32c⟩
⟨Rand RAM 69b⟩
⟨Message RAM 28⟩
⟨Semaphore RAM 25⟩
⟨Task Stack RAM 35a⟩


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; area configuration
; we want absolute dataspace, with this area called "CODE"
.area   .CODE (ABS)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RST functions

; RST 00
⟨RST 00 implementation 22⟩

; RST 08
⟨RST 08 implementation 23a⟩

; RST 10
⟨RST 10 implementation 23b⟩

; RST 18
⟨RST 18 implementation 23c⟩

; RST 20
⟨RST 20 implementation 23d⟩

; RST 28
⟨RST 28 implementation 24a⟩

; RST 30
⟨RST 30 implementation 24b⟩

; RST 38
⟨RST 38 implementation 24c⟩
```

```
; NMI
⟨NMI implementation 24d⟩



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; interrupt service routine:
⟨Interrupt Service Routine implementation 31a⟩


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; the core OS stuff:
    ; initialization and splash screen
⟨.start implementation 13a⟩

    ; the core task
⟨.coretask implementation 46⟩


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; some helpful utility functions

; memset256
⟨Utils memset256 implementation 66⟩


; memsetN
⟨Utils memsetN implementation 67⟩


; clear screen
⟨Utils cls implementation 68⟩

; clear screen (gui tile version)
⟨Utils guicls implementation 69a⟩


; rand
⟨Utils rand implementation 70b⟩


; sine
⟨Utils sine implementation 71⟩

; cosine
⟨Utils cosine implementation 72⟩


; text justification
⟨Utils textcenter implementation 75⟩
```

```
⟨Utils textright implementation 76⟩


; xy2offs
⟨Utils xy2offsB implementation 82⟩

⟨Utils xy2offsAC implementation 79⟩


; putstr
⟨Utils putstrA implementation 80⟩

⟨Utils putstrB implementation 85⟩

⟨Utils putstrC implementation 81⟩



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; semaphore control

; lock semaphore
⟨Semaphore lock implementation 26⟩

; release semaphore
⟨Semaphore release implementation 27⟩


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; task exec, kill, and sleep routines

⟨Exec start implementation 49⟩

⟨Exec kill implementation 50⟩


⟨Exec sleep implementation 51⟩


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The tasks

; task list -- list of all available tasks
⟨Task List 39a⟩


;;;;;;;;;;;;;;;;;;;;
; task number 0
⟨Task 0 implementation 58b⟩
```

```
;;;;;;;;;;;;;;;;;;;;
; task number 1
⟨Task 1 implementation 60a⟩


;;;;;;;;;;;;;;;;;;;;
; task number 2
⟨Task 2 implementation 62a⟩


;;;;;;;;;;;;;;;;;;;;
; task number 3
⟨Task 3 implementation 64a⟩


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The Data

; splash strings
⟨Init splash data 17a⟩

; Some tables for the Task Switcher
⟨Task Switch ROM 35b⟩

; The sine table
⟨Utils sine table 73⟩

; The XY-offset table
⟨Utils scroffs table 83⟩

; The Region A and C offset table
⟨Utils acoffs table 78⟩
```

This code is used in chunk 100.

Appendix D

Auxiliary Data Files

This chapter defines all of the extra files needed to convert the generated ASM as well as the auxiliary PCX image files into the ROM files that we need to generate.

The two types of files, `.ROMS` and `.INI` are needed for the external **`genroms`** and **`turacoCL`** programs, which are used to generate the ROM images.

D.1 genroms .ROMS files

These files are the data files used by “genroms” to produce ROM image files from the generated Intel Hex File (.IHX) by the makefile.

The basic fields are:

- start address
- rom size
- rom filename
- rom reference name

D.1.1 Ms. Pac-Man

106 ⟨*mspacman.roms* 106⟩≡

```
# program space
begin program
0x0000  0x1000  boot1   program_1
0x1000  0x1000  boot2   program_2
0x2000  0x1000  boot3   program_3
0x3000  0x1000  boot4   program_4
0x8000  0x1000  boot5   program_5
```

```
0x9000  0x1000  boot6   program_6
end

# graphics bank 1
begin graphics
0x0000  0x1000  5e      graphics_1

# graphics bank 2
0x0000  0x1000  5f      graphics_2
end

# color proms
begin color
0x0000  0x0020  82s123.7f       palette
0x0020  0x0100  82s126.4a       colorlookup
end

# sound proms
begin sound
0x0000  0x0100  82s126.1m       sound_a
0x0100  0x0100  82s126.3m       sound_timing
end
```

Root chunk (not used in this document).

D.1.2 Pac-Man

108 ⟨*pacman.roms* 108⟩≡

```
# program space
begin program
0x0000  0x1000  pacman.6e       program_1
0x1000  0x1000  pacman.6f       program_2
0x2000  0x1000  pacman.6h       program_3
0x3000  0x1000  pacman.6j       program_4
end

# graphics bank 1
begin graphics
0x0000  0x1000  pacman.5e       graphics_1

# graphics bank 2
0x0000  0x1000  pacman.5f       graphics_2
end

# color proms
begin color
0x0000  0x0020  82s123.7f       palette
0x0020  0x0100  82s126.4a       colorlookup
end

# sound proms
begin sound
0x0000  0x0100  82s126.1m       sound_a
0x0100  0x0100  82s126.3m       sound_timing
end
```

Root chunk (not used in this document).

D.1.3 Pengo 2u

109 ⟨*pengo2u.roms* 109⟩≡

```
begin program
0x0000  0x1000  pengo.u8            program_1
0x1000  0x1000  pengo.u7            program_2
0x2000  0x1000  pengo.u15           program_3
0x3000  0x1000  pengo.u14           program_4
0x4000  0x1000  pengo.u21           program_5
0x5000  0x1000  pengo.u20           program_6
0x6000  0x1000  pengo.u32           program_7
0x7000  0x1000  pengo.u31           program_8
end

# graphics bank 1
begin graphics
0x0000  0x2000  ic92           graphics_1

# graphics bank 2
0x0000  0x2000  ic105          graphics_2
end

# color and palette proms proms
begin color
0x0000  0x0020  pr1633.078     palette
0x0020  0x0400  pr1634.088     colorlookup
end

# sound proms
begin sound
0x0000  0x0100  pr1635.051     sound_a
0x0100  0x0100  pr1636.070     sound_timing
end
```

Root chunk (not used in this document).

D.2 turaco .INI file

These files are used to convert the .pcx files into graphics ROM image files by "turacoCL". The exact format of this file will not be described here since it is outside of the scope of this document.

For more detail about what is going on here, please refer to the documentation and sample .ini driver contained in the "turacoCL" package.

D.2.1 (Ms.) Pac-Man

110 ⟨*pacman.ini* 110⟩≡

```
[Turaco]
FileVersion = 1.0
DumpVersion = 2
Author = Jerry / MAME 0.65.1 Dump
URL = http://www.cis.rit.edu/~jerry/Software/turacoCL

[General]
Name = pacman
Grouping = pacman
Year = 1980
Manufacturer = [Namco] (Midway license)
CloneOf = puckman
Description = Pac-Man (Midway)

[Layout]
GfxDecodes = 2

[GraphicsRoms]
Rom1 =      0    4096  pacman.5e
Rom2 =   4096    4096  pacman.5f

[Decode1]
start = 0
width = 8
height = 8
total = 256
orientation = 0
planes = 2
planeoffsets = 0 4
xoffsets = 56 48 40 32 24 16 8 0
yoffsets = 64 65 66 67 0 1 2 3
charincrement = 128

[Decode2]
start = 4096
width = 16
height = 16
```

```
total = 64
planes = 2
planeoffsets = 0 4
xoffsets = 312 304 296 288 280 272 264 256 56 48 40 32 24 16 8 0
yoffsets = 64 65 66 67 128 129 130 131 192 193 194 195 0 1 2 3
charincrement = 512

[Palette]
Palette1  = 4    0  0  0   220 220 220     0   0  90   220   0   0
Palette2  = 4    0  0  0     0 220   0     0   0  90   220 150  20
Palette3  = 4    0  0  0     0   0 220   255   0   0   255 255   0
Palette4  = 4    0  0  0   220   0   0    90  90   0   220 220 220
Palette5  = 4    0  0  0   220   0   0     0 220   0   220 220 220
Palette6  = 4    0  0  0   150 150   0     0 220   0    90  90   0
Palette7  = 4    0  0  0   220 220   0    90  90 220   220 220 220
Palette8  = 4    0  0  0   220   0   0    90  90   0   220 220 220
Palette9  = 4    0  0  0     0 150 220     0 220   0   220 220 220
Palette10 = 4    0  0  0     0   0   0    90  90 220   220 220 220
Palette11 = 4  255  0  0   255 255 255     0 255   0     0   0 220
Palette12 = 4    0  0  0   255 255 255     0   0   0     0   0 220
```

Root chunk (not used in this document).

D.2.2 Pengo

112 $\langle$*pengo2u.ini* 112$\rangle\equiv$

```
[General]
Description = Pengo (set 2 not encrypted)

[Layout]
GfxDecodes = 4
Orientation = 5

[GraphicsRoms]
Rom1 = 0 8192 ic92
Rom2 = 8192 8192 ic105

[Decode1]
start = 0
width = 8
height = 8
total = 256
planes = 2
planeoffsets = 0 4
xoffsets = 64 65 66 67 0 1 2 3
yoffsets = 0 8 16 24 32 40 48 56
charincrement = 128

[Decode2]
start = 4096
width = 16
height = 16
total = 64
planes = 2
planeoffsets = 0 4
xoffsets = 64 65 66 67 128 129 130 131 192 193 194 195 0 1 2 3
yoffsets = 0 8 16 24 32 40 48 56 256 264 272 280 288 296 304 312
charincrement = 512

[Decode3]
start = 8192
width = 8
height = 8
total = 256
planes = 2
planeoffsets = 0 4
xoffsets = 64 65 66 67 0 1 2 3
yoffsets = 0 8 16 24 32 40 48 56
charincrement = 128

[Decode4]
start = 12288
width = 16
```

```
height = 16
total = 64
planes = 2
planeoffsets = 0 4
xoffsets = 64 65 66 67 128 129 130 131 192 193 194 195 0 1 2 3
yoffsets = 0 8 16 24 32 40 48 56 256 264 272 280 288 296 304 312
charincrement = 512

[Palette]
Palette1  = 4    0  0  0   220 220 220     0   0  90   220   0   0
Palette2  = 4    0  0  0     0 220   0     0   0  90   220 150  20
Palette3  = 4    0  0  0     0   0 220   255   0   0   255 255   0
```

Root chunk (not used in this document).

Appendix E

Building Alpaca

This chapter explains what is necessary to build ALPACA, as well as how to do so.

E.1 Required software

To start off with, you will need some software packages installed to build anything:

To do anything:

- gnu make (gmake)
- noweb/notangle
- unix tools: cat, cd, cp, dd, uname, zip

To build the document:

- ImageMagick tools: convert
- LaTeX / PDFLaTeX

To build the romset:

- genroms
- turaco CL
- ZCC package or asz80 and aslink

To test the romset:

- MAME or some other emulator

E.2 Makefile targets

Once you have the correct software installed, as explained in the previous section, you should just be able to type "gmake"[1] and have it build this document docs/alpaca-development.pdf as well as the rom image files as specified in the makefile. See below on how to specify Pac-Man or Pengo roms.

As a side effect, a well commented Z80 ASM file will be in "code/alpaca.asm" for your viewing pleasure. To make things a little easier to see, you might want to do a `make listing` to generate the "code/alpaca.lst" listing file.

In a nutshell, you can just type `make targetname` to make that specific target's files. The valid targets are:

paclisting builds: code/pacalpaca.lst listing file

pacprog builds: code/pacalpaca.asm, code/pacfinal.ihx

pacroms builds: roms/pacman/* (graphics and code)

pacromzip builds a zip of the above roms

pactest builds the above roms, runs MAME to test them out

pengolisting builds: code/pengoalpaca.lst listing file

pengoprog builds: code/pengoalpaca.asm, code/pengofinal.ihx

pengoroms builds: roms/pengo2u/* (graphics and code)

pengoromzip builds a zip of the above roms

pengotest builds the above roms, runs MAME to test them out

docs builds: doc/alpaca.pdf

dview builds: doc/alpaca.pdf, runs acroread

clean gets rid of all targets

tidy cleans the doc directory of intermediate files

all builds: doc/alpaca.pdf, code/pacalpaca.asm, code/pacalpaca.lst, code/pengoalpaca.asm, code/pengoalpaca.lst, pac and pengo rom image files into roms/

dist builds: "all", then puts it in a new directory

backup builds a .tar.gz file of the whole source tree

You may need to change the paths to the MAME program and ROM directories in the makefile if you want to run the test targets on your system.

[1] or "make" on OS X

E.3 The Makefile

116 ⟨*GNUmakefile* 116⟩≡

```
# GNUMakefile for the Alpaca project
#
#    Scott "Jerry" Lawrence
#
#  It's not pretty.  Sorry about that.
#

#
# $Id: build.nw,v 1.9 2003/08/14 14:51:55 jerry Exp $
#

############################################################
# Targets:
#       paclisting      builds: code/pacalpaca.lst listing file
#       pacprog         builds: code/pacalpaca.asm, code/pacfinal.ihx
#       pacroms         builds: roms/pacman/pacman.* (graphics and code)
#       pacromzip       builds a zip of the above roms
#       pactest         builds the pac-man roms, runs MAME to test them out
#
#       pengolisting    builds: code/pacalpaca.lst listing file
#       pengoprog       builds: code/pacalpaca.asm, code/pacfinal.ihx
#       pengoroms       builds: roms/pengo/pengo.* (graphics and code)
#       pengoromzip     builds a zip of the above roms
#       pengotest       builds the pengo roms, runs MAME to test them out
#
#       docs            builds: doc/alpaca.pdf
#       dview           builds: doc/alpaca.pdf, runs acroread
#
#       clean           gets rid of all targets
#       tidy            cleans the doc directory of intermediate files
#
#       dist            web-ready distribution
#       backup          source distribution (everything)
#
#       all             builds: docs, roms, listing

############################################################
all: docs paclisting pengolisting pacroms pengoroms
############################################################
test: paclisting pactest
############################################################

HAS_NOWEB := 1

############################################################

# program name
```

```
PROG    := alpaca
VERSION := 0.7

# extra programs
GENROMS  := genroms
TURACOCL := turacocl
DD       := dd
ZIP      := zip
TAR      := tar --exclude=CVS --exclude=.*
BLDSYS   := $(shell uname -s)

# directories
CODEDIR   := code
ROMSROOT  := roms
ROMSOURCE := roms/dummy
DISTDIR   := $(PROG)_$(VERSION)

# backup files
THISDIR   := alpaca
TARFILE   := $(PROG)_$(VERSION)_src.tar


#########################################
# emulator selection
#   - for testing romsets

# if we want to use xmame on OS X, set EMULATOR to ForceXMame
EMULATOR := ForceXMame


# the name of the xmame executable
XMAME    := xmame

# the name of the xmame executable with the debugger compiled in
XMAMED   := xmamed -debug

# parameters for all Xmame versions:
MAMEPARAMS := -skip_disclaimer -skip_gameinfo

# and the xmame to use.  (set XMAMED to XMAME for no debugger)
XMAMEUSE := $(XMAME) $(MAMEPARAMS)


# apps and dirs for OS X testing of Pac-Man

# osx app to use to test Pac roms
PMTAPP := /Applications/jerry/Games/MacPacMAME\ 0.58/MacPacMAME\ 0.58
# dir to copy pac roms into
PMTRD  := /Applications/jerry/Games/MacPacMAME\ 0.58/ROMS/pengman
```

```
# apps and dirs for OS X testing of Pengo

# osx app to use to test Pengo roms
PGTAPP := /Applications/jerry/Games/MacMAME/MacMAME.app
# dir to copy pengo roms into
PGTRD  := /Applications/jerry/Games/MacMAME/ROMs/pengo2u


########################################

ifdef HAS_NOWEB

NWS := \
	nws/title.nw \
	nws/overview.nw \
	nws/arch.nw \
	nws/init.nw \
	nws/kernserv.nw \
	nws/semaphores.nw \
	nws/messages.nw \
	nws/malloc.nw \
	nws/isr.nw \
	nws/coretask.nw \
	nws/exec.nw \
	nws/task0.nw \
	nws/task1.nw \
	nws/task2.nw \
	nws/task3.nw \
	nws/utils.nw \
	nws/error.nw \
	\
	nws/appendix.nw \
	nws/schedule.nw \
	nws/hardware.nw \
	nws/asm.nw \
	nws/auxdata.nw \
	nws/build.nw \
	nws/license.nw \
	nws/end.nw

PCX :=\
	gfx/pacscreen.pcx \
	gfx/pac_1.pcx \
	gfx/pac_1c.pcx \
	gfx/pac_2.pcx \
	gfx/pac_2c.pcx

PCXPDF := $(PCX:%.pcx=%.pdf)

endif
```

```
STYLE := doc/alpaca.sty
DOC := doc/$(PROG).pdf

docs:   $(DOC)

dview:  docs
        open $(DOC)

############################################################

PACTARG := $(CODEDIR)/pacfinal.ihx
PACASMS := $(CODEDIR)/pacalpaca.asm

PENGOTARG := $(CODEDIR)/pengofinal.ihx
PENGOASMS := $(CODEDIR)/pengoalpaca.asm

DEPS :=

DATA :=

CLEAN := Release Build $(DISTDIR)

ifdef HAS_NOWEB
    CLEAN += $(PENGOTARG) $(PENGOTARG:%.ihx=%.map)
    CLEAN += $(PENGOASMS) $(PENGOASMS:%.asm=%.rel)
    CLEAN += $(PACTARG) $(PACTARG:%.ihx=%.map)
    CLEAN += $(PACASMS) $(PACASMS:%.asm=%.rel)
    CLEAN += doc/alpaca* code/*.lst
    CLEAN += roms/pacman/pacman.* pac*.zip
    CLEAN += roms/pengo2u/pengo*.* pengo*.zip
    CLEAN += roms/pengo2u/ic*
endif

TIDY := $(COMMON_OBJS) $(STYLE) \
        $(DOC:%.pdf=%.tex) $(DOC:%.pdf=%.aux) \
        $(DOC:%.pdf=%.log) $(DOC:%.pdf=%.toc) \
        $(PCXPDF) $(DOC:%.pdf=%.out)



############################################################
# Pac builds

# various config
PACROMDIR      := $(ROMSROOT)/pacman
PACBACKDIR     := ../..
PACGENROMSFILE := $(CODEDIR)/pacman.roms
PACTURACOINI   := $(CODEDIR)/pacman.ini
PACROMNAME     := pacman
```

```
CLEAN += $(PACGENROMSFILE)
CLEAN += $(PACTURACOINI)


pacprog:        $(PACTARG)
.PHONY: pacprog

pacroms:        $(PACTARG) $(PACGENROMSFILE) $(PACTURACOINI)
        cd $(PACROMDIR) ;\
                $(GENROMS) $(PACBACKDIR)/$(PACGENROMSFILE)\
                           $(PACBACKDIR)/$(PACTARG)
        $(DD) if=/dev/zero of=$(PACROMDIR)/pacman.5e bs=4096 count=1
        $(DD) if=/dev/zero of=$(PACROMDIR)/pacman.5f bs=4096 count=1
        $(TURACOCL) -inf IMG -bnk 1 -rod $(PACROMDIR)\
                    -rom $(PACROMDIR) -ini $(PACTURACOINI)\
                    -dbf gfx/pac_1.pcx
        $(TURACOCL) -inf IMG -bnk 2 -rod $(PACROMDIR)\
                    -rom $(PACROMDIR) -ini $(PACTURACOINI)\
                    -dbf gfx/pac_2.pcx
.PHONY: pacroms

pacromzip:      pacroms
        mkdir $(PACROMNAME)
        cp $(PACROMDIR)/8* $(PACROMDIR)/p* $(PACROMNAME)
        $(ZIP) -r $(PACROMNAME).zip $(PACROMNAME)
        rm -rf $(PACROMNAME)
.PHONY: pacromzip


########################################
# PAC test targets

# automagically choose the correct one..
ifeq ($(BLDSYS),Darwin)
 ifeq ($(EMULATOR),ForceXMame)
pactest:        pacroms mamepactest
 else
pactest:        pacroms osxpactest
 endif
else
pactest:        pacroms mamepactest
endif
.PHONY: pactest


osxpactest:
        cp -f $(PACROMDIR)/pacman.* $(PMTRD)
        cp -f $(PACROMDIR)/82*.* $(PMTRD)
        open -a $(PMTAPP)
```

```
.PHONY: osxpactest


mamepactest:
	$(XMAMEUSE) -rp $(ROMSROOT) pacman
.PHONY: mamepactest



############################################################
# Pengo builds

# various config
PENGOROMDIR      := $(ROMSROOT)/pengo2u
PENGOBACKDIR     := ../..
PENGOGENROMSFILE := $(CODEDIR)/pengo2u.roms
PENGOTURACOINI   := $(CODEDIR)/pengo2u.ini
PENGOROMNAME     := pengo2u

CLEAN += $(PENGOGENROMSFILE)
CLEAN += $(PENGOTURACOINI)


pengoprog:	$(PENGOTARG)
.PHONY: pengoprog

pengoroms:	$(PENGOTARG) $(PENGOGENROMSFILE) $(PENGOTURACOINI)
	cd $(PENGOROMDIR) ;\
		$(GENROMS) $(PENGOBACKDIR)/$(PENGOGENROMSFILE)\
		$(PENGOBACKDIR)/$(PENGOTARG)
	$(DD) if=/dev/zero of=$(PENGOROMDIR)/ic92 bs=8192 count=1
	$(DD) if=/dev/zero of=$(PENGOROMDIR)/ic105 bs=8192 count=1
	$(TURACOCL) -inf IMG -bnk 1 -rod $(PENGOROMDIR)\
	            -rom $(PENGOROMDIR) -ini $(PENGOTURACOINI)\
	            -dbf gfx/pen_1.pcx
	$(TURACOCL) -inf IMG -bnk 2 -rod $(PENGOROMDIR)\
	            -rom $(PENGOROMDIR) -ini $(PENGOTURACOINI)\
	            -dbf gfx/pen_2.pcx
	$(TURACOCL) -inf IMG -bnk 3 -rod $(PENGOROMDIR)\
	            -rom $(PENGOROMDIR) -ini $(PENGOTURACOINI)\
	            -dbf gfx/pen_3.pcx
	$(TURACOCL) -inf IMG -bnk 4 -rod $(PENGOROMDIR)\
	            -rom $(PENGOROMDIR) -ini $(PENGOTURACOINI)\
	            -dbf gfx/pen_4.pcx
.PHONY: pengoroms

pengoromzip:	pengoroms
	mkdir $(PENGOROMNAME)
	cp $(PENGOROMDIR)/ic* $(PENGOROMDIR)/p* $(PENGOROMNAME)
	$(ZIP) -r $(PENGOROMNAME).zip $(PENGOROMNAME)
```

```
	rm -rf $(PENGOROMNAME)
.PHONY: pengoromzip


########################################
# PENGO test targets

# automagically choose the correct one..
ifeq ($(BLDSYS),Darwin)
 ifeq ($(EMULATOR),ForceXMame)
pengotest:	pengoroms mamepengotest
 else
pengotest:	pengoroms osxpengotest
 endif
else
pengotest:	pengoroms mamepengotest
endif
.PHONY: pengotest

osxpengotest:
	cp -f $(PENGOROMDIR)/pengo.* $(PGTRD)
	cp -f $(PENGOROMDIR)/ic* $(PGTRD)
	cp -f $(PENGOROMDIR)/pr163*.* $(PGTRD)
	open -a $(PGTAPP)
.PHONY: osxpengotest


mamepengotest:
	$(XMAMEUSE) -rp $(ROMSROOT) pengo2u
.PHONY: mamepengotest


############################################################

clean:	tidy
	rm -rf $(CLEAN)

tidy:
	rm -rf $(TIDY)

dist: docs paclisting pacromzip pengolisting pengoromzip
	rm -rf $(DISTDIR)
	mkdir $(DISTDIR)
	cp $(DOC) $(DISTDIR)
	cp $(PACLSTS) $(PACASMS) $(DISTDIR)
	cp $(PACROMNAME).zip $(DISTDIR)
	cp $(PENGOLSTS) $(PENGOASMS) $(DISTDIR)
	cp $(PENGOROMNAME).zip $(DISTDIR)

backup: clean
```

```
        cd ..; $(TAR) -cvf $(TARFILE) $(THISDIR)
        gzip -f ../$(TARFILE)


############################################################


PACRELS         := $(PACASMS:%.asm=%.rel)
PACLSTS         := $(PACASMS:%.asm=%.lst)

PENGORELS       := $(PENGOASMS:%.asm=%.rel)
PENGOLSTS       := $(PENGOASMS:%.asm=%.lst)

paclisting:     $(PACLSTS)
pengolisting:   $(PENGOLSTS)

%.lst: %.asm
        asz80 -l $<
.SECONDARY: $(PACASMS) $(PENGOASMS)

OPTS    := -O

$(PACTARG): $(PACRELS)
        aslink -i -m -o $(PACTARG) -b_CODE=0x0000 $(PACRELS)

$(PENGOTARG): $(PENGORELS)
        aslink -i -m -o $(PENGOTARG) -b_CODE=0x0000 $(PENGORELS)

%.rel: %.asm
        asz80 $<

%.rel: %.c
        zcc -c -v $(OPTS) -D$(ARCH) -D$(TEST) -I../include $(ADDS) $<

.SECONDARY: $(PACTARG)
.SECONDARY: $(PENGOTARG)

############################################################

ifdef HAS_NOWEB

$(CODEDIR)/%.asm:       $(NWS)
        -@$(MKDIR_CMD)
        notangle -R$*.asm $^ | cpif $@

$(CODEDIR)/%.roms:      $(NWS)
        -@$(MKDIR_CMD)
        notangle -R$*.roms $^ | cpif $@

$(CODEDIR)/%.ini:       $(NWS)
```

```
        -@$(MKDIR_CMD)
        notangle -R$*.ini $^ | cpif $@


%.pdf:  %.tex
        -@$(MKDIR_CMD)
        ( \
            cd $(@D); \
            oldFingerprint="ZZZ" ; \
            if [ -f $*.aux ]; then \
                fingerprint="`sum $*.aux`" ; \
            else \
                fingerprint="YYY" ; \
            fi ; \
            while [ ! "$${oldFingerprint}" = "$${fingerprint}" ]; do \
                oldFingerprint="$${fingerprint}" ; \
                pdflatex $(<F) ; \
                fingerprint="`sum $(*F).aux`" ; \
            done ; \
        )

$(DOC:%.pdf=%.tex):     $(PCXPDF) $(NWS)
        -@$(MKDIR_CMD)
        cat $(NWS) | noweave -delay -index | cpif $@

doc/%.sty: nws/%.sty
        -@$(MKDIR_CMD)
        cp $< $@


%.pdf: %.pcx
        convert $< $@

endif

############################################################

.PHONY: all
.PHONY: docs
.PHONY: clean
.PHONY: tidy

#.SECONDARY: $(TIDY)

############################################################

$(DOC): $(PCXPDF) $(STYLE)

############################################################
```

Root chunk (not used in this document).

Appendix F

Software License

This software, “Alpaca” is covered by the GNU Lesser General Public License. The terms of this license are covered as follows:

F.1 The Short Version

125 ⟨*license short version* 125⟩≡

```
;;   Alpaca - A Multitasking operating system for Z80 arcade hardware
;;   Copyright (C) 2003 Scott "Jerry" Lawrence
;;                      alpaca@umlautllama.com
;;
;;    This is free software; you can redistribute it and/or modify
;;    it under the terms of the GNU Lesser General Public License
;;    as published by the Free Software Foundation; either version
;;    2 of the License, or (at your option) any later version.
;;
;;    This software is distributed in the hope that it will be
;;    useful, but WITHOUT ANY WARRANTY; without even the implied
;;    warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
;;    PURPOSE.  See the GNU Lesser General Public License for
;;    more details.
;;
;;    You should have received a copy of the GNU Lesser General
;;    Public License along with this library; if not, write to
;;    the Free Foundation, Inc., 59 Temple Place, Suite 330,
;;    Boston, MA  02111-1307  USA
```

This code is used in chunk 101.

F.2 The Long Version

126 ⟨*license long version* 126⟩≡

```
                     GNU LESSER GENERAL PUBLIC LICENSE
                         Version 2.1, February 1999

  Copyright (C) 1991, 1999 Free Software Foundation, Inc.
      59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
  Everyone is permitted to copy and distribute verbatim copies
  of this license document, but changing it is not allowed.

 [This is the first released version of the Lesser GPL.  It also counts
  as the successor of the GNU Library Public License, version 2, hence
  the version number 2.1.]

                                Preamble

   The licenses for most software are designed to take away your
 freedom to share and change it.  By contrast, the GNU General Public
 Licenses are intended to guarantee your freedom to share and change
 free software--to make sure the software is free for all its users.

   This license, the Lesser General Public License, applies to some
 specially designated software packages--typically libraries--of the
 Free Software Foundation and other authors who decide to use it.  You
 can use it too, but we suggest you first think carefully about whether
 this license or the ordinary General Public License is the better
 strategy to use in any particular case, based on the explanations below.

   When we speak of free software, we are referring to freedom of use,
 not price.  Our General Public Licenses are designed to make sure that
 you have the freedom to distribute copies of free software (and charge
 for this service if you wish); that you receive source code or can get
 it if you want it; that you can change the software and use pieces of
 it in new free programs; and that you are informed that you can do
 these things.

   To protect your rights, we need to make restrictions that forbid
 distributors to deny you these rights or to ask you to surrender these
 rights.  These restrictions translate to certain responsibilities for
 you if you distribute copies of the library or if you modify it.

   For example, if you distribute copies of the library, whether gratis
 or for a fee, you must give the recipients all the rights that we gave
 you.  You must make sure that they, too, receive or can get the source
 code.  If you link other code with the library, you must provide
 complete object files to the recipients, so that they can relink them
 with the library after making changes to the library and recompiling
 it.  And you must show them these terms so they know their rights.
```

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free

```
programs enables a greater number of people to use a large body of
free software.  For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

  Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.

  The precise terms and conditions for copying, distribution and
modification follow.  Pay close attention to the difference between a
"work based on the library" and a "work that uses the library".  The
former contains code derived from the library, whereas the latter must
be combined with the library in order to run.

                  GNU LESSER GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License Agreement applies to any software library or other
program which contains a notice placed by the copyright holder or
other authorized party saying it may be distributed under the terms of
this Lesser General Public License (also called "this License").
Each licensee is addressed as "you".

  A "library" means a collection of software functions and/or data
prepared so as to be conveniently linked with application programs
(which use some of those functions and data) to form executables.

  The "Library", below, refers to any such software library or work
which has been distributed under these terms.  A "work based on the
Library" means either the Library or any derivative work under
copyright law: that is to say, a work containing the Library or a
portion of it, either verbatim or with modifications and/or translated
straightforwardly into another language.  (Hereinafter, translation is
included without limitation in the term "modification".)

  "Source code" for a work means the preferred form of the work for
making modifications to it.  For a library, complete source code means
all the source code for all modules it contains, plus any associated
interface definition files, plus the scripts used to control compilation
and installation of the library.

  Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running a program using the Library is not restricted, and output from
such a program is covered only if its contents constitute a work based
on the Library (independent of the use of the Library in a tool for
writing it).  Whether that is true depends on what the Library does
```

and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN

Root chunk (not used in this document).