# Adventures in Writing an Operating System

**K. Isom**

**Nov 09, 2023**

# CONTENTS:

CHAPTER

ONE

# CHAPTER 1: INTRODUCTION

I've wanted to write an general purpose operating system for a long time - it's one of my computing bucketlist items. kOS, pronounced "chaos", is my project to do that. Along the way, I'm going to write up my work[1] to help me understand what's going on.

kOS will be a 64-bit operating system.

## 1.1 The first target

The first target is the Raspberry Pi 4B; mine has 8GB of memory available. It has a Broadcom BCM2711 System on Chip (SoC), which features a Cortex A72 quad-core ARMv8 processor running at 1.5 GHz. The L1 cache is 32kB for data, 48kB for instructions, and the L2 cache is 1MB. It also has a Videocore VI GPU, which has built-in support for OpenGL ES 3.1 and Vulkan 1.2.

As for I/O, it has a PCIe bus, a DSI[2] and CSI[3] bus[4], support for up to six I2C buses, up to six UARTs[5], and up to five SPI buses[6]. There are also a pair of HDMI outputs.

The BCM2711 has 58 general purpose I/O lines; 27 of these are exposed via the 40-pin header on the Raspberry Pi.

Some of the relevant pinouts:

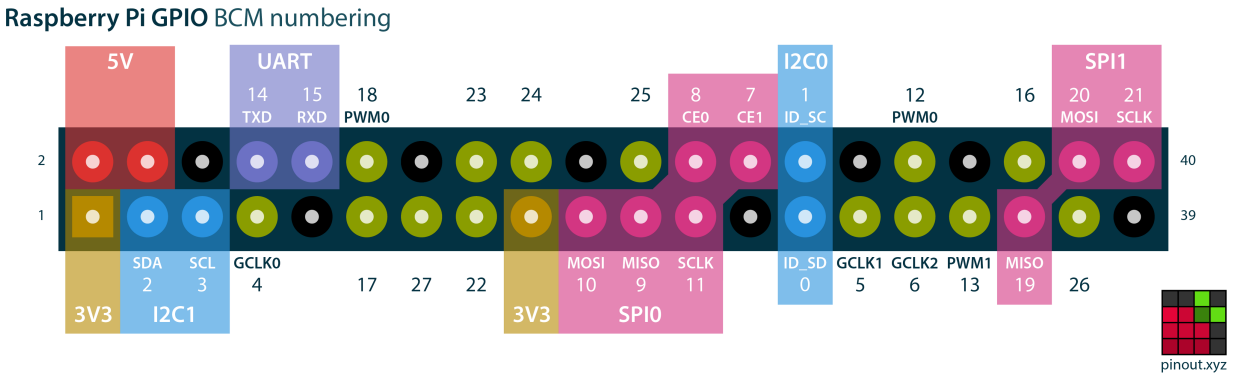


Fig. 1: 40-pin header pinout diagram, courtesy of pinout.xyz.

[1] Or at least try to… no guarantees I'll keep up on it.
[2] Display Serial Interface.
[3] Camera Serial Interface.
[4] Technically, the processor has two DSI and two CSI buses, but there are only one of each exposed on the Raspberry Pi.
[5] Universal Asynchronous Receiver Transmitter, aka serial ports.
[6] Again, technically there are six, but only five are exposed on the Raspberry Pi.

## 1.2 Booting the target

The Raspberry Pi documentation describes the boot sequence; there is an SPI flash EEPROM that has the initial bootloader.

1. The SoC powers up.
2. If the nRPIBOOT GPIO is high, or if the OTP hasn't configured an nRPIBOOT pin[7], try to load `recovery.bin`[8] and update the SPI EEPROM.
3. Try to load the second-stage bootloader from the SPI EEPROM.
4. Initialize SDRAM and clocks.
5. Read the EEPROM configuration files ().
6. Check to see if a HALT is requested.
7. Read the next boot mode from the `BOOT_ORDER` parameter in the EEPROM configuration. The boot modes are:
    - `RESTART`: jump back to the first boot mode in the parameter.
    - `STOP`: display the start.elf error pattern (see the appendices for a description of these) and loop indefinitely.
    - `SD_CARD`: try to load from the SD card.
    - `NETWORK`: use DHCP to get an IP address, then use either DHCP or TFTP to get a boot image.
    - `USB-MSD`: check for USB mass storage, and try to load firmware from each `LUN` discovered.
    - `NVME`: the same, but using NVMe.
    - `RPIBOOT`: trying to load firmware from a USB device connected to the USB-OTG port.

The first partition on the SD card must be a FAT partition, from which the bootloader looks for certain files:

- Prior to the Pi 4, `bootcode.bin` is loaded first, which then loads a `start.elf` variant. The Pi 4 uses its SPI EEPROM instead.
- The `start.elf` variants are firmware. The `start4` versions are used for the Raspberry Pi. - `start.elf`: basic firmware. - `start_x.elf`: has additional codecs. - `start_db.elf`: used for debugging. - `start_cd.elf`: cut-down firmware, removing hardware blocks.
- The `start.elf` firmware is paired with an appropriately-named linker file, `fixup.dat`.
- `config.txt`: RPi configuration.
- `overlays/`: device-tree overlays

The minimum we'll need to bring up the kernel are

- `bcm2711-rpi-4-b.dtb`
- `config.txt`
- `fixup4.dat`
- `start4.elf`

Finally, our `kernel8.img` will go in the partition.

[7] The docs point out that only the CM4 configures an nRPIBOOT pin.
[8] `recovery.bin` contains a minimal second-stage bootloader to reflash the SPI EEPROM.

## 1.3 Resources

- The OSDev Wiki
- BCM2711 ARM Peripherals
- ARM Cortex-A72 MPCore Processor Technical Reference Manual

## 1.4 Footnotes

CHAPTER

TWO

# CHAPTER 2: BUILD INFRASTRUCTURE

## 2.1 aarch64-none-elf toolchain

If you've done other bare-metal ARM projects, you might be used to the `arm-none-eabi` toolchain. For this project, we'll need the arm64 equivalent: `aarch64-none-elf`. You'll probably need to fetch this from the ARM toolchain download page and add it to your `PATH`. I unpacked it into `~/.local/aarch64-none-elf`, and added the `bin` dir to my `PATH`.

## 2.2 Makefile

Later on, this might get more complicated; for now, a short Makefile will suffice. The code will be laid out as

```
.
├── build
├── inc
└── src
```

First, we'll set up some basic tooling and variables:

```
## VARIABLES
#  B -      build dir
#  D -      disk (SD) image
#  H -      header file / include dir
#  I -      output image (place this on μSD card)
#  L -      linker script
#  M -      map file
#  P -      path to sources
#  S -      assembler listing
#  T -      toolchain
B ?=                build
D ?=                $(B)/sd.img
H ?=                inc
I ?=                $(B)/kernel8.img
L ?=                pi4.ld
M ?=                $(B)/kernel8.map
P ?=                src
S ?=                $(B)/kernel8.list
T ?=                aarch64-none-elf
```

(continues on next page)

(continued from previous page)

```
PLATFORM :=         pi4
ELF :=              $(B)/kospi64.elf

AS :=               $(T)-as
CC :=               $(T)-gcc
CXX :=              $(T)-g++
LD :=               $(T)-ld
OC :=               $(T)-objcopy
OD :=               $(T)-objdump
```

The build flags will be fairly standard using compile flags targeting the Cortex A72; we will be providing out own startfile and standard library.

```
BUILD_FLAGS := -O2 -Wall -Werror -ffreestanding -march=armv8-a+crc  \
               -nostartfiles -nostdinc -nostdlib                    \
               -Wno-unused-command-line-argument -I$(H)
LDFLAGS :=  -nostdlib --no-undefined
CFLAGS :=   $(BUILD_FLAGS)
CXXFLAGS := -std=c++17 $(BUILD_FLAGS)
```

Next, collect all the source files and collect them into a variable describing the objects that need to be built.

```
SRCXX :=            $(wildcard $(P)/*.cc)
SRCCC :=            $(wildcard $(P)/*.c)
SRCAS :=            $(wildcard $(P)/*.S)
OBJS :=             $(patsubst $(P)/%.S,$(B)/%.o,$(SRCAS))
OBJS +=             $(patsubst $(P)/%.cc,$(B)/%.o,$(SRCXX))
OBJS +=             $(patsubst $(P)/%.c,$(B)/%.o,$(SRCCC))
```

Later, we'll use podman to build and test this with an image I've prepared with the development tooling needed to cross-compile.

```
PODMAN ?=   podman
PIMAGE ?=   git.wntrmute.dev/kyle/armdev:latest
```

Make's default target should be the kernel image we're building.

```
all:        $(I)
```

The `list` target produces an assembly language listing of the kernel.

```
list:       $(S)

$(S):       $(ELF)
        $(OD) -d $(ELF) > $@
```

The contents of the ELF will be dumped into a memory dump. All of its symbols and relocation information are discarded.

```
$(I):       $(ELF)
        $(OC) $(ELF) -O binary $@
```

The ELF file is the linked output of all the object files; the build should also produce a remapping list. In order to build it, we need a linker map describing how memory on the Raspberry Pi is laid out. The object files are built in a pretty

standard way.

```
$(ELF):     $(OBJS) $(L)
        $(LD) -o $@ $(LDFLAGS) $(OBJS) -Map $(M) -T $(L)

$(B)/%.o: $(P)/%.S $(B)
        $(CC) -o $@ $(BUILD_FLAGS) -c -I $(P) $<

$(B)/%.o: $(P)/%.cc $(B)
        $(CXX) -o $@ -c $(CXXFLAGS) -I $(P) $<

$(B)/%.o: $(P)/%.c $(B)
        $(CC) -o $@ -c $(CFLAGS) -I $(P) $<

$(B):
        mkdir $@
```

Finally, we have a few utility targets. Right now, the disk image isn't actually being used for anything — later on, I'd like to get qemu emulation support working. The `print-` target shows the value of whatever variable is supplied; e.g.

```
% make print-OBJS
OBJS= build/main.o
```

The devbox targets use `podman` to build a development image for the kernel. The `devdeps` target assumes a Debian-like system.

```
$(D): $(B)
        dd if=/dev/zero of=$@ bs=128k count=8192
        mkfs.msdos -n boot $@

print-%: ; @echo '$(subst ','\'',$*=$($*))'

clean :
        -rm -rf $(B) $(I) $(S) $(M) $(D)
devbox-build:
        $(PODMAN) build -t $(PIMAGE) -f Containerfile

devbox-run:
        $(PODMAN) run -i -t -v $(PWD):/build $(PIMAGE)

devdeps:
        sudo apt install binutils-arm-none-eabi gcc-arm-none-eabi   \
                libnewlib-arm-none-eabi libstdc++-arm-none-eabi-dev \
                libstdc++-arm-none-eabi-newlib                      \
                libstdc++-arm-none-eabi-picolibc picolibc-arm-none-eabi

.PHONY: all list clean emulate emulate-vga devbox-build devbox-run devdeps
```

The next step is to supply the bare minimum to build the kernel:

- a main souce file
- an assembly-language bootloader
- a linker script

CHAPTER

THREE

# CHAPTER 3: BASE-BONES SCAFFOLDING

## 3.1 kmain.cc

The easiest piece of this component is our kernel basic code. At first, all we need it to do is spin forever. This will go in `src/kmain.cc`.

We need to make the `kmain` name visible to the bootloader; C++ name mangling means that otherwise, `kmain` will be linked as something like `_Z5kmainv`. Rather than trying to to figure out what name will be and making sure that name is known to the bootloader, we can just tell the linker to turn off mangling for this function using `extern "C"`.

```
extern "C" {
    void kmain();
}

void
kmain()
{
        while(true) ;
}
```

## 3.2 boot.S

The bootloader starts at the beginning of the text segment.

```
.section ".text.boot"

.globl _start

_start:
```

To make things easier, we're going to tell the last three cores to halt. To do that, we need need to query the control processor to figure out which processor we are. There's only four processors, so we AND by 3 to clear out any other bits. Then, if the processor isn't #0, jump on over to the halt label.

This is discussed on page 4-92 of [CA72TRM].

```
mrs         x1,     mpidr_el1
and         x1,     x1, #3
cmp         x1,     #0
bne         halt
```

Now, set up our stack at 0x80000, which is the address the kernel will be loaded into in memory. This is set by the Raspberry Pi bootloader.

```
mov         sp,     #0x80000
```

The base segment (.bss) is where any statically allocated variables that have been declared, but not initialized. So, we should initialize them. We do that by starting at the beginning of the BSS section, and continuing until we've iterated `__bss_size` times.

```
        ldr         x1,     =__bss_start
        ldr         w2,     =__bss_size
zero_bss:
```

`str` is the mnemonic for Store Register. We store the contents of the `xzr`[1] register into the address pointed to by `x1` - which we initialized with the start of the BSS section. We add 8 to `x1` — 64 bits — and subtract one from the size of the BSS. If this is not zero, we keep going. If it is, we'll branch with link (`bl`) to the label `kmain`. The `bl` mnemonic tells the assembler that this is probably calling some routine. It is an unconditional jump, and the expectation here is that we never return.

```
str         xzr,    [x1],   #8
sub         w2,     w2,     #1
cbnz        w2,     zero_bss

bl kmain
```

The halt block loops through an endless loop, waiting for events (`wfe`).

```
halt:
        wfe
        b halt
```

## 3.3 The linker script

This is honestly taken almost directly from the Raspberry Pi forums. I'll take a stab at explaining it, and hope that this forces me to understand it. This script has three parts to it: a pointer to the entry point (e.g. where in `boot.S` it should begin execution), a block describing what memory is avilable and how much of it there is, and then a block describing the sections of memory.

We start by setting the entrypoint to the `_start` symbol defined previously.

```
ENTRY(_start)
```

The memory region definition follows. Quoting from the documentation[2]:

```
The syntax for 'MEMORY' is:
 MEMORY
   {
     NAME [(ATTR)] : ORIGIN = ORIGIN, LENGTH = LEN
     ...
   }
```

(continues on next page)

[1] The xzr register is a pseudo-register that always returns zero.
[2] This is from the info page, e.g. `info ld`, section 3.7.

(continued from previous page)

```
The NAME is a name used in the linker script to refer to the region.
The region name has no meaning outside of the linker script.
```

I don't know if the memory section is strictly necessary here; the reference in the SECTIONS block could be replaced with a hardcoded number. The Raspberry Pi bootloader firmware (in the EEPROM) loads the kernel from memory location 0x80000, so that'll get marked as the load address. I have 8 GB of RAM in the Pi 4, but for the sake of starting the kernel, I'll mark the lowest standard amount of memory, which I *think* is 1GB. We also shouldn't need to write to this memory, so it's marked as read-execute only.

```
MEMORY
{
        LOAD (rx)  : ORIGIN = 0x0000080000, LENGTH = 1g
}
```

Now we need to define our memory sections; we previously talked about one of them (BSS), but there are some others we'll need to set up. The guiding principles here are to put symbols in the same region if they need to be initialized or if there's a specific reason they need to be grouped together.

BSS is one such grouping: static variables that need to be initialized. Another grouping are static variables that have been initialized and must be loaded from memory. On microcontrollers (e.g. Cortex-M series systems), that might be from flash. This will be loaded from the SD card image, e.g. from SDRAM. On those systems, you would also pay attention to making sure constant data and code lives in read-only memory. This matters when you don't have a lot of RAM, but for kOS, I'm not going to worry about that. We do have to mark certain areas for heaps. The common ones I've seen is

- `.text` for code,
- `.bss` for uninitialized data,
- `.stack` for the stack, and
- `.data` for initialized data.

It's worth noting that there's a spec; appendix 1 covers the reserved names.

```
SECTIONS
{
```

The current point in memory, aka the first block of memory, is set to the LOAD memory address from our previous definitions. Every section that follows continues from here. The `KEEP` directive ensures that the `text.boot` section keeps the that particular section at the beginning. Since our bootloader (`boot.S`) starts at `text.boot`, we want that to be where our memory starts. The `linkonce` directive says it should be linked in only once.

```
. = LOAD;
.text :
{
        KEEP(*(.text.boot))
                *(.text .text.* .gnu.linkonce.t*)
}

.rodata :
{
        *(.rodata .rodata.* .gnu.linkonce.r*)
}
```

(continues on next page)

(continued from previous page)

```
PROVIDE(_data = .);

.data :
{
        *(.data
        .data.*
        .gnu.linkonce.d*)
}
```

The `bss` section is reserved, but there's nothing to load from memory (because we are going to initialize it to zero). This section is accordingly marked as `NOLOAD`. It needs to be aligned to 16-bytes as per the spec.

```
        .bss (NOLOAD) :
        {
                . = ALIGN(16);
                __bss_start = .;
                *(.bss .bss.*)
                *(COMMON)
                __bss_end = .;
        }
        _end = .;

        /DISCARD/ : { *(.comment) *(.gnu*) *(.note*) *(.eh_frame*) }
}
```

Finlly, we define a `__bss_size` that we'll use when initializing the BSS.

```
__bss_size = (__bss_end - __bss_start)>>3;
```

After building `kospi64.elf`, we can use `objdump` to view the sections - I've shortened the program header addresses to make them fit, but they are 64-bit addresses.

```
kyle@midgard:~/src/kospi64$ make
mkdir build
aarch64-none-elf-g++ -o build/boot.o -O2 -Wall -Werror -ffreestanding
>-march=armv8-a+crc -nostartfiles -nostdinc -nostdlib -Wno-unused-
>command-line-argument -Iinc -c -I src src/boot.S
aarch64-none-elf-g++ -o build/main.o -c -std=c++17 -O2 -Wall -Werror
>-ffreestanding -march=armv8-a+crc -nostartfiles -nostdinc -nostdlib
>-Wno-unused-command-line-argument -Iinc -I src src/main.cc
aarch64-none-elf-ld -o build/kospi64.elf -nostdlib --no-undefined
>build/boot.o build/main.o -Map build/kernel8.map -T pi4.ld
aarch64-none-elf-objcopy build/kospi64.elf -O binary build/kernel8.img
kyle@midgard:~/src/kospi64$ aarch64-none-elf-objdump -x build/kospi64.elf

build/kospi64.elf:     file format elf64-littleaarch64
build/kospi64.elf
architecture: aarch64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000008000

Program Header:
    LOAD off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**16
```

(continues on next page)

(continued from previous page)

```
         filesz 0x00008044 memsz 0x00008044 flags r-x
    LOAD off    0x00000000 vaddr 0x1f000000 paddr 0x1f000000 align 2**16
         filesz 0x00000000 memsz 0x00000000 flags rw-
private flags = 0x0:

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000044  0000000000008000  0000000000008000  00008000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .bss          00000000  000000001f000000  000000001f000000  00010000  2**0
                  ALLOC
SYMBOL TABLE:
0000000000008000 l    d  .text      0000000000000000 .text
000000001f000000 l    d  .bss       0000000000000000 .bss
0000000000000000 l    df *ABS*      0000000000000000 boot.o
000000000000802c l       .text      0000000000000000 halt
000000000000801c l       .text      0000000000000000 zero_bss
0000000000000000 l    df *ABS*      0000000000000000 main.cc
0000000000000000 g       *ABS*      0000000000000000 __bss_size
000000001f000000 g       .bss       0000000000000000 __bss_end
0000000000008000 g       .text      0000000000000000 _start
000000001f000000 g       .bss       0000000000000000 __bss_start
000000001f000000 g       .bss       0000000000000000 _end
0000000000008040 g     F .text      0000000000000004 kmain


kyle@midgard:~/src/kospi64$
```

## 3.4 The future

This will be enough to bootstrap a simple kernel. Later on, we'll want to load a kernel from external media, whether the SD card, an SSD, or NVMe drive. With that in mind, the definitions above should be enough to get this basic functionaltiy working (in particularly the memory is overkill).

At this point we have a boot kernel, but no way to tell that it's booting (except maybe with a JTAG probe), so the next step should be getting a serial console working.

## 3.5 Footnotes

CHAPTER

FOUR

# CHAPTER 4: FIRST STEPS: THE UART

The standard UART on the Raspberry Pi are pins 8 (GPIO14) and 10 (GPIO15). In their first alternate mode (ALT0), these become TXD0 and RXD0. TXD0 should be connected to the RX pin on the UART cable, and RXD0 to the TX pin.

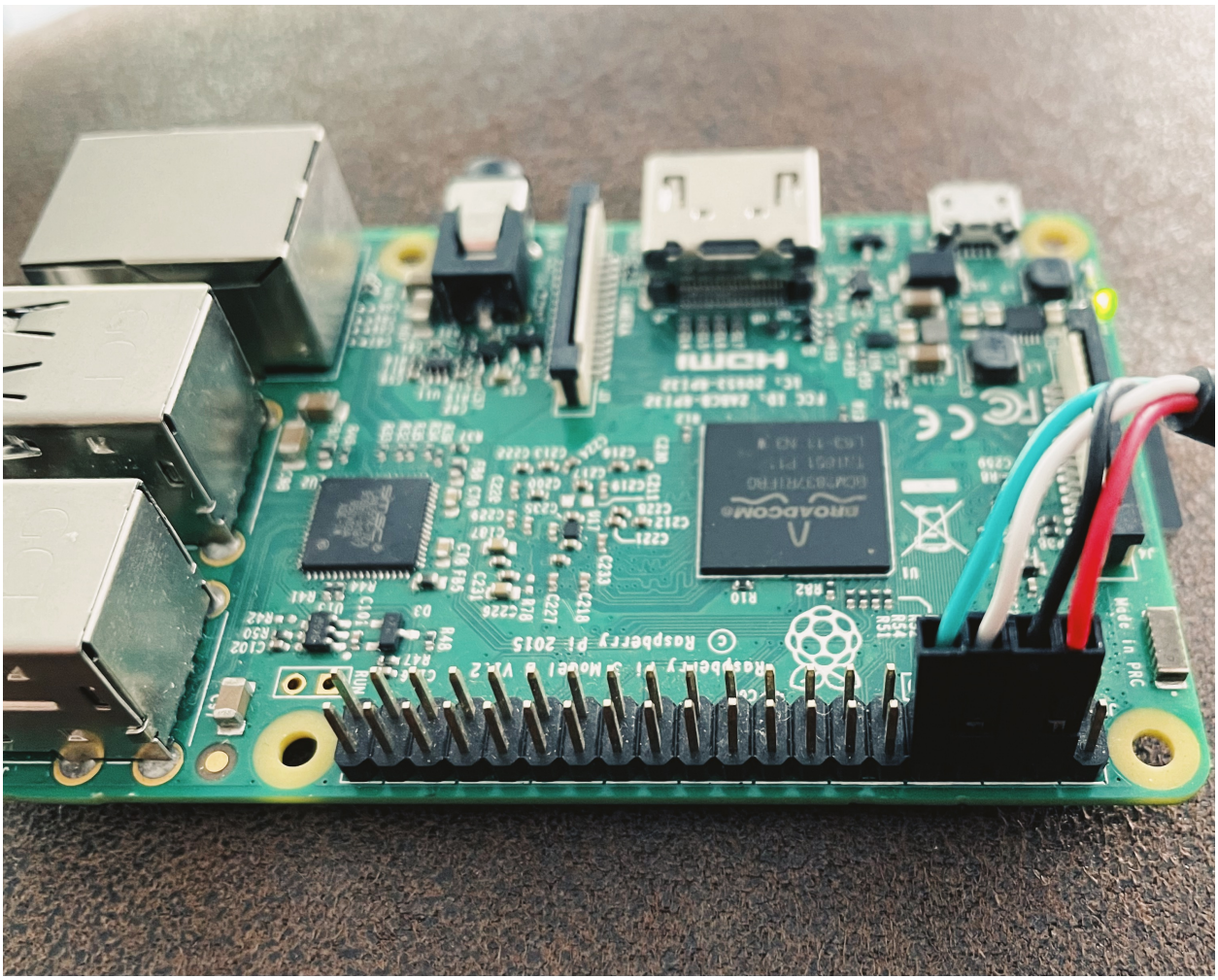

Fig. 1: The cable has the standard Raspberry Pi serial console color scheme: green is TX, white is RX, black is ground, and red is 5V/500mA. Note that the Raspberry Pi 4 won't work with voltage input.

CHAPTER

FIVE

# APPENDIX A: LED ERROR PATTERNS

| Long flashes | Short flashes | Error |
|---|---|---|
| 0 | 3 | Generic failure to boot |
| 0 | 4 | start*.elf not found |
| 0 | 7 | **Kernel image not found** |
| 0 | 8 | SDRAM failure |
| 0 | 9 | Insufficient SDRAM |
| 0 | 10 | In HALT state |
| 2 | 1 | Partition not FAT |
| 2 | 2 | Failed to read from partition |
| 2 | 3 | Extended partition not FAT |
| 2 | 4 | File signature/hash mismatch |
| 3 | 1 | SPI EEPROM error |
| 3 | 2 | SPI EEPROM write-protected |
| 3 | 3 | I2C error |
| 3 | 4 | Secure boot config invalid |
| 4 | 4 | Unsupported board type |
| 4 | 5 | Fatal firmware error |
| 4 | 6 | Power failure type A |
| 4 | 7 | Power failure type B |

CHAPTER

SIX

# APPENDIX B: THE KOS MAKEFILE

```
#### KYOSPI/64 MAKE OF GREAT SUCCESS

## VARIABLES
#  B -      build dir
#  D -      disk (SD) image
#  H -      header file / include dir
#  I -      output image (place this on μSD card)
#  L -      linker script
#  M -      map file
#  P -      path to sources
#  S -      assembler listing
#  T -      toolchain
B ?=                build
D ?=                $(B)/sd.img
# i shake my first at the tyranny that doesn't let me name this
# kyospi64.img.
H ?=                inc
I ?=                $(B)/kernel8.img
L ?=                pi4.ld
M ?=                $(B)/kernel8.map
P ?=                src
S ?=                $(B)/kernel8.list
T ?=                arm-none-eabi

ELF :=              $(B)/kyospi64.elf

AS :=               $(T)-as
CC :=               $(T)-gcc
CXX :=              $(T)-g++
LD :=               $(T)-ld
OC :=               $(T)-objcopy
OD :=               $(T)-objdump

BUILD_FLAGS := -O2 -Wall -Werror -ffreestanding -march=armv8-a+crc  \
               -mfloat-abi=hard -mfpu=crypto-neon-fp-armv8           \
               -nostartfiles -nostdinc -nostdlib                    \
               -Wno-unused-command-line-argument -I$(H)
LDFLAGS :=  -nostdlib --no-undefined
CFLAGS :=   $(BUILD_FLAGS)
CXXFLAGS := -std=c++17 $(BUILD_FLAGS)
```

(continues on next page)

(continued from previous page)

```
SRCXX :=    $(wildcard $(P)/*.cc)
SRCCC :=    $(wildcard $(P)/*.c)
OBJS :=     $(patsubst $(P)/%.S,$(B)/%.o,$(wildcard $(P)/*.S))
OBJS +=     $(patsubst $(P)/%.cc,$(B)/%.o,$(SRCXX))
OBJS +=     $(patsubst $(P)/%.c,$(B)/%.o,$(SRCCC))

PODMAN ?=   podman # can also be docker if you hate yourself
PIMAGE ?=   git.wntrmute.dev/kyle/armdev:latest


all:        $(I)

$(S):       $(ELF)
        $(OD) -d $(ELF) > $@

$(I):       $(ELF)
        $(OC) $(ELF) -O binary $@

$(ELF):     $(OBJS) $(L)
        $(LD) -o $@ $(LDFLAGS) $(OBJS) -Map $(M) -T $(L)

$(B)/%.o: $(P)/%.S $(B)
        $(CC) -o $@ $(BUILD_FLAGS) -c -I $(P) $<

$(B)/%.o: $(P)/%.cc $(B)
        $(CXX) -o $@ -c $(CXXFLAGS) -I $(P) $<

$(B)/%.o: $(P)/%.c $(B)
        $(CC) -o $@ -c $(CFLAGS) -I $(P) $<

$(B):
        mkdir $@

$(D): $(B)
        dd if=/dev/zero of=$@ bs=128k count=8192

print-%: ; @echo '$(subst ','\'',$*=$($*))'

clean :
        -rm -rf $(B) $(I) $(S) $(M) $(D)

emulate: $(I) $(D)
        qemu-system-aarch64                                     \
                -M raspi3b                                      \
                -cpu cortex-a53                                 \
                -m 1G                                           \
                -smp 4                                          \
                -kernel $(I)                                    \
                -serial telnet:localhost:4321,server,nowait     \
                -monitor telnet:localhost:4322,server,nowait    \
                -device usb-net,netdev=net0                     \
```

(continues on next page)

(continued from previous page)

```
                -nographic                                                      \
                -drive format=raw,file=$(D),if=sd,media=disk

emulate-vga: $(I) $(D)
        qemu-system-aarch64                                                     \
                -M raspi3b                                                      \
                -cpu cortex-a53                                                 \
                -m 1G                                                           \
                -smp 4                                                          \
                -device VGA,id=vga2                                             \
                -kernel $(I)                                                    \
                -serial telnet:localhost:4321,server,nowait                     \
                -monitor telnet:localhost:4322,server,nowait                    \
                -device usb-net,netdev=net0                                     \
                -drive format=raw,file=$(D),if=sd,media=disk

devbox-build:
        $(PODMAN) build -t $(PIMAGE) -f Containerfile

devbox-run:
        $(PODMAN) run -i -t -v $(PWD):/build $(PIMAGE)

.PHONY: all clean
```

CHAPTER

SEVEN

# BIBLIOGRAPHY


- [CA72TRM] ARM. ARM® Cortex®-A72 MPCore Processor Technical Reference Manual, Revision r0p1. 2014.
- [BCM2711AP] Raspberry Pi Ltd. BCM2711 ARM Peripherals, 2022. Build date: 2022-01-18. Build qversion: githash: cfcff44-clean.

CHAPTER

EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search