Raspberry Pi Bare Bones

From OSDev Wiki

This is a tutorial on operating systems development on the Raspberry Pi. This tutorial is written specifically for the Raspberry Pi Model B Rev 2 because the author has no other hardware to test on. But so far the models are basically identical for the purpose of this tutorial (Rev 1 has 256MB ram, Model A has no ethernet). This will serve as an example of how to create a minimal system, but not as an example of how to properly structure your project.

Difficulty level

🟩⬜⬜⬜
Beginner

Kernel Designs

Models

Monolithic Kernel
Microkernel
Hybrid Kernel
Exokernel
Nano/Picokernel
Cache Kernel
Virtualizing Kernel
Megalithic Kernel

Other Concepts

Modular Kernel
Higher Half Kernel
64-bit Kernel

WAIT! Have you read Getting Started, Beginner Mistakes, and some of the related OS theory?

Contents

Prepare

You are about to begin development of a new operating system. Perhaps one day, your new operating system can be developed under itself. This is a process known as bootstrapping or going self-hosted. However, that is way into the future. Today, we simply need to set up a system that can compile your operating system from an existing operating system. This is a process known as cross-compiling and this makes the first step in operating systems development.

This article assumes you are using a Unix-like operating system such as Linux which supports operating systems development well. Windows users should be able to complete it from a MinGW or Cygwin environment.

Building a Cross-Compiler

Main article: GCC Cross-Compiler, Why do I need a Cross Compiler?

The first thing you should do is set up a GCC Cross-Compiler for **arm-none-eabi**. You have not yet modified your compiler to know about the existence of your operating system, so we use a generic target called arm-none-eabi, which provides you with a toolchain targeting the System V ABI. You will *not* be able to correctly compile your operating system without a cross-compiler.

Overview

By now, you should have set up your cross-compiler for arm-none-eabi (as described above). This tutorial provides a minimal solution for creating an operating system. It doesn't serve as a recommend skeleton for project structure, but rather as an example of a minimal kernel. In this simple case, we just need three input files:

- boot.S - kernel entry point that sets up the processor environment
- kernel.c - your actual kernel routines
- linker.ld - for linking the above files

Booting the Operating System

We will now create a file called boot.S and discuss its contents. In this example, we are using the GNU assembler, which is part of the cross-compiler toolchain you built earlier. This assembler integrates very well with the rest of the GNU toolchain.

```
// To keep this in the first portion of the binary.
.section ".text.boot"

// Make _start global.
.globl _start

// Entry point for the kernel.
// r15 -> should begin execution at 0x8000.
// r0 -> 0x00000000
// r1 -> 0x00000C42
// r2 -> 0x00000100 - start of ATAGS
// preserve these registers as argument for kernel_main
_start:
        // Setup the stack.
        mov sp, #0x8000

        // Clear out bss.
        ldr r4, =__bss_start
        ldr r9, =__bss_end
        mov r5, #0
        mov r6, #0
        mov r7, #0
        mov r8, #0
        b       2f

1:
        // store multiple at r4.
        stmia r4!, {r5-r8}
        // If we are still below bss_end, loop.
2:
        cmp r4, r9
        blo 1b

        // Call kernel_main
        ldr r3, =kernel_main
        blx r3

        // halt
halt:
        wfe
        b halt
```

The section ".text.boot" will be used in the linker script to place the boot.S as the very first thing in our kernel image. The code initializes a minimum C environment, which means having a stack and zeroing the BSS segment, before calling the kernel_main function. Note that the code avoids using r0-r2 so the remain valid for the kernel_main call.

You can then assemble boot.S using:

```
arm-none-eabi-gcc -mcpu=arm1176jzf-s -fpic -ffreestanding -c boot.S -o bc
```

Implementing the Kernel

So far we have written the bootstrap assembly stub that sets up the processor such that high level languages such as C can be used. It is also possible to use other languages such as C++.

Freestanding and Hosted Environments

If you have done C or C++ programming in user-space, you have used a so-called Hosted Environment. Hosted means that there is a C standard library and other useful runtime features. Alternatively, there is the Freestanding version, which is what we are using here. Freestanding means that there is no C standard library, only what we provide ourselves. However, some header files are actually not part of the C standard library, but rather the compiler. These remain available even in freestanding C source code. In this case we use <stdbool.h> to get the bool datatype, <stddef.h> to get size_t and NULL, and <stdint.h> to get the intx_t and uintx_t datatypes which are invaluable for operating systems development, where you need to make sure that the variable is of an exact size (if we used a short instead of uint16_t and the size of short changed, our VGA driver here would break!). Additionally you can access the <float.h>, <iso646.h>, <limits.h>, and <stdarg.h> headers, as they are also freestanding. GCC actually ships a few more headers, but these are special purpose.

Writing a kernel in C

The following shows how to create a simple kernel in C. Please take a few moments to understand the code.

```c
#if !defined(__cplusplus)
#include <stdbool.h>
#endif
#include <stddef.h>
#include <stdint.h>

static inline void mmio_write(uint32_t reg, uint32_t data)
{
        *(volatile uint32_t *)reg = data;
}

static inline uint32_t mmio_read(uint32_t reg)
{
        return *(volatile uint32_t *)reg;
}

/* Loop <delay> times in a way that the compiler won't optimize away. */
static inline void delay(int32_t count)
{
        asm volatile("__delay_%=: subs %[count], %[count], #1; bne __dela
                 : : [count]"r"(count) : "cc");
}

size_t strlen(const char* str)
{
        size_t ret = 0;
        while ( str[ret] != 0 )
                ret++;
        return ret;
}

enum
{
    // The GPIO registers base address.
    GPIO_BASE = 0x20200000,

    // The offsets for reach register.

    // Controls actuation of pull up/down to ALL GPIO pins.
    GPPUD = (GPIO_BASE + 0x94),

    // Controls actuation of pull up/down for specific GPIO pin.
    GPPUDCLK0 = (GPIO_BASE + 0x98),

    // The base address for UART.
    UART0_BASE = 0x20201000,

    // The offsets for reach register for the UART.
    UART0_DR     = (UART0_BASE + 0x00),
    UART0_RSRECR = (UART0_BASE + 0x04),
    UART0_FR     = (UART0_BASE + 0x18),
    UART0_ILPR   = (UART0_BASE + 0x20),
    UART0_IBRD   = (UART0_BASE + 0x24),
    UART0_FBRD   = (UART0_BASE + 0x28),
    UART0_LCRH   = (UART0_BASE + 0x2C),
    UART0_CR     = (UART0_BASE + 0x30),
    UART0_IFLS   = (UART0_BASE + 0x34),
    UART0_IMSC   = (UART0_BASE + 0x38),
    UART0_RIS    = (UART0_BASE + 0x3C),
    UART0_MIS    = (UART0_BASE + 0x40),
    UART0_ICR    = (UART0_BASE + 0x44),
    UART0_DMACR  = (UART0_BASE + 0x48),
    UART0_ITCR   = (UART0_BASE + 0x80),
    UART0_ITIP   = (UART0_BASE + 0x84),
    UART0_ITOP   = (UART0_BASE + 0x88),
    UART0_TDR    = (UART0_BASE + 0x8C),
};

void uart_init()
{
        // Disable UART0.
        mmio_write(UART0_CR, 0x00000000);
        // Setup the GPIO pin 14 && 15.

        // Disable pull up/down for all GPIO pins & delay for 150 cycles.
        mmio_write(GPPUD, 0x00000000);
        delay(150);

        // Disable pull up/down for pin 14,15 & delay for 150 cycles.
        mmio_write(GPPUDCLK0, (1 << 14) | (1 << 15));
        delay(150);

        // Write 0 to GPPUDCLK0 to make it take effect.
        mmio_write(GPPUDCLK0, 0x00000000);

        // Clear pending interrupts.
        mmio_write(UART0_ICR, 0x7FF);

        // Set integer & fractional part of baud rate.
        // Divider = UART_CLOCK/(16 * Baud)
        // Fraction part register = (Fractional part * 64) + 0.5
        // UART_CLOCK = 3000000; Baud = 115200.

        // Divider = 3000000 / (16 * 115200) = 1.627 = ~1.
        // Fractional part register = (.627 * 64) + 0.5 = 40.6 = ~40.
        mmio_write(UART0_IBRD, 1);
        mmio_write(UART0_FBRD, 40);

        // Enable FIFO & 8 bit data transmissio (1 stop bit, no parity).
        mmio_write(UART0_LCRH, (1 << 4) | (1 << 5) | (1 << 6));
        // Mask all interrupts.
        mmio_write(UART0_IMSC, (1 << 1) | (1 << 4) | (1 << 5) | (1 << 6)
                               (1 << 7) | (1 << 8) | (1 << 9) | (1 << 10)

        // Enable UART0, receive & transfer part of UART.
        mmio_write(UART0_CR, (1 << 0) | (1 << 8) | (1 << 9));
}

void uart_putc(unsigned char byte)
{
        // Wait for UART to become ready to transmit.
        while ( mmio_read(UART0_FR) & (1 << 5) ) { }
        mmio_write(UART0_DR, byte);
}

unsigned char uart_getc()
{
    // Wait for UART to have recieved something.
    while ( mmio_read(UART0_FR) & (1 << 4) ) { }
    return mmio_read(UART0_DR);
}

void uart_write(const unsigned char* buffer, size_t size)
{
        for ( size_t i = 0; i < size; i++ )
                uart_putc(buffer[i]);
}

void uart_puts(const char* str)
{
        uart_write((const unsigned char*) str, strlen(str));
}

#if defined(__cplusplus)
extern "C" /* Use C linkage for kernel_main. */
#endif
void kernel_main(uint32_t r0, uint32_t r1, uint32_t atags)
{
        (void) r0;
        (void) r1;
        (void) atags;

        uart_init();
        uart_puts("Hello, kernel World!\r\n");

        while ( true )
                uart_putc(uart_getc());
}
```

The GPU bootloader passes arguments to the kernel via r0-r2 and the boot.S makes sure to preserve those 3 registers. They are the first 3 arguments in a C function call. The argument r0 contains a code for the device the rpi was booted from. This is generally 0 but its actual value depends on the firmware of the board. r1 contains the 'ARM Linux Machine Type' which for the rpi is 3138 (0xc42) identifying the bcm2708 cpu. A full list of ARM Machine Types is available from here (http://www.arm.linux.org.uk/developer/machines/) . r2 contains the address of the ATAGs.

Notice how we wish to use the common C function strlen, but this function is part of the C standard library that we don't have available. Instead, we rely on the freestanding header <stddef.h> to provide size_t and we simply declare our own implementation of strlen. You will have to do this for every function you wish to use (as the freestanding headers only provide macros and data types).

Compile using:

```
arm-none-eabi-gcc -mcpu=arm1176jzf-s -fpic -ffreestanding -std=gnu99 -c k
```

Note that the above code uses a few extensions and hence we build as the GNU version of C99.

Linking the Kernel

To create the full and final kernel we will have to link these object files into the final kernel program. When developing user-space programs, your toolchain ships with default scripts for linking such programs. However, these are unsuitable for kernel development and we need to provide our own customized linker script.

```
ENTRY(_start)

SECTIONS
{
    /* Starts at LOADER_ADDR. */
    . = 0x8000;
    __start = .;
    __text_start = .;
    .text :
    {
        KEEP(*(.text.boot))
        *(.text)
    }
    . = ALIGN(4096); /* align to page size */
    __text_end = .;

    __rodata_start = .;
    .rodata :
    {
        *(.rodata)
    }
    . = ALIGN(4096); /* align to page size */
    __rodata_end = .;
    __data_start = .;
    .data :
    {
        *(.data)
    }
    . = ALIGN(4096); /* align to page size */
    __data_end = .;

    __bss_start = .;
    .bss :
    {
        bss = .;
        *(.bss)
    }
    . = ALIGN(4096); /* align to page size */
    __bss_end = .;
    __end = .;
}
```

There is a lot of text here but don't despair. The script is rather simple if you look at it bit by bit.

ENTRY(_start) declares the entry point for the kernel image. That symbol was declared in the boot.S file. Since we are actually booting a binary image, the entry is completely irrelevant, but it has to be there in the elf file we build as intermediate file.

SECTIONS declares sections. It decides where the bits and pieces of our code and data go and also sets a few symbols that help us track the size of each section.

```
    . = 0x8000;
    __start = .;
```

The "." denotes the current address so the first line tells the linker to set the current address to 0x8000, where the kernel starts. The current address is automatically incremented when the linker adds data. The second line then creates a symbol "__start" and sets it to the current address.

After that sections are defined for text (code), read-only data, read-write data and BSS (0 initialized memory). Other than the name the sections are identical so lets just look at one of them:

```
    __text_start = .;
    .text : {
        KEEP(*(.text.boot))
        *(.text)
    }
    . = ALIGN(4096); /* align to page size */
    __text_end = .;
```

The first line creates a __text_start symbol for the section. The second line opens a .text section for the output file which gets closed in the fifth line. Lines 3 and 4 declare what sections from the input files will be placed inside the output .text section. In our case ".text.boot" is to be placed first followed by the more general ".text". ".text.boot" is only used in boot.S and ensures that it ends up at the beginning of the kernel image. ".text" then contains all the remaining code. Any data added by the linker automatically increments the current addrress ("."). In line 6 we explicitly increment it so that it is aligned to a 4096 byte boundary (which is the page size for the RPi). And last line 7 creates a __text_end symbol so we know where the section ends.

What are the __text_start and __text_end for and why use page alignment? The 2 symbols can be used in the kernel source and the linker will then place the correct addresses into the binary. As an example the __bss_start and __bss_end are used in boot.S. But you can also use the symbols from C by declaring them extern first. While not required I made all sections aligned to page size. This later allows mapping them in the page tables with executable, read-only and read-write permissions without having to handle overlaps (2 sections in one page).

```
    __end = .;
```

After all sections are declared the __end symbol is created. If you ever want to know how large your kernel is at runtime you can use __start and __end to find out.

With these components you can now actually build the final kernel. We use the compiler as the linker as it allows it greater control over the link process. Note that if your kernel is written in C++, you should use the C++ compiler instead.

You can then link your kernel using:

```
arm-none-eabi-gcc -T linker.ld -o myos.elf -ffreestanding -O2 -nostdlib b
arm-none-eabi-objcopy myos.elf -O binary myos.bin
```

Note: This kernel isn't currently linking with libgcc as Bare Bones is and this may be a mistake.

Booting the Kernel

In a few moments, you will see your kernel in action.

Testing your operating system (Real Hardware)

Do you still have the SD card with the original Raspbian image on it from when you where testing the hardware above? Great. So you already have a SD card with a boot partition and the required files. If not then download one of the original raspberry boot images and copy them to the SD card.

Now mount the first partition from the SD card and look at it:

```
bootcode.bin  fixup.dat     kernel.img           start.elf
cmdline.txt   fixup_cd.dat  kernel_cutdown.img   start_cd.elf
```

```
config.txt    issue.txt     kernel_emergency.img
```

Simplified when the RPi powers up the ARM cpu is halted and the GPU runs. The GPU loads the bootloader from ROM and executes it. That then finds the SD card and loads the bootcode.bin. The bootcode handles the config.txt and cmdline.txt (or does start.elf read that?) and then runs start.elf. start.elf loads the kernel.img and at last the ARM cpu is started running that kernel image.

So now we replace the original kernel.img with our own, umount, sync, stick the SD card into RPi and turn the power on. Your minicom should then show the following:

```
Hello World
```

Testing your operating system (QEMU)

Although vanilla QEMU does not yet support the RaspberryPi hardware, there is a fork by Torlus (https://github.com/Torlus/qemu/tree/rpi) that emulates the RPi hardware sufficiently enough to get started with ARM kernel development. To build QEMU on linux, do as follows:

```
mkdir qemu-build
cd qemu-build
git clone https://github.com/Torlus/qemu/tree/rpi src
mkdir build
cd build
../src/configure --prefix=$YOURINSTALLLOCATION --target-list=arm-softmmu,
make && sudo make install
```

With qemu you do not need to objcopy the kernel into a plain binary; QEMU also supports ELF kernels:

```
$YOURINSTALLLOCATION/bin/qemu-system-arm -kernel kernel.elf -cpu arm1176
```

Note that currently the QEMU "raspi" emulation may incorrectly load the kernel binaries at 0x10000 instead of 0x8000, so if you do not see any output, try adjusting the base address constant in the linker script.

See Also

External Links

- BCM2835 ARM Peripherals (http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf)