Framebuffer, Pitch, and Depth

A **Framebuffer** is a piece of memory that is shared between the CPU and the GPU. The CPU writes RGB pixels to the buffer, and the GPU renders it to whatever output device you have connected.

The **Depth** of a framebuffer is the number of bits in every pixel. For this tutorial, we will be exclusively using a depth of 24, meaning that each of the red, green and blue values will have 8 bits, or 1 byte to them.

The **Pitch** of a framebuffer is simply the number of bytes that are in each row on screen.

We can calculate the **Pixels Per Row** by `pitch / (depth / 8)`, or equivalently `(pitch * 8) / depth`.

We can calculate the offset within the framebuffer of a pixel located at **Coordinates (x,y)** by `pitch * y + (depth / 8) * x`

Jake Sandler
jsandler18@gmail.com
jsandler18

Interrupts and Exceptions

Exceptions

An **Exception** is an event that is triggered when something exceptional occurs during normal program execution. Examples of such exceptional occurrences include hardware devices presenting new data to the CPU, user code asking to perform a privileged action, and a bad instruction was encountered.

On the Raspberry Pi, when an exception occurs, a specific address is loaded into the program counter register, branching execution to this point. At this location, the kernel developer needs to write branch instructions to routines that handle the exceptions. This set of addresses, also known as the **Vector Table**, starts at address 0. Below is a table that describes each exception

Address	Exception Name	Exception Source	Action to take
0x00	Reset	Hardware Reset	Restart the Kernel
0x04	Undefined instruction	Attempted to execute a meaningless instruction	Kill the offending program
0x08	Software Interrupt (SWI)	Software wants to execute a privileged operation	Perform the opertation and return to the caller
0x0C	Prefetch Abort	Bad memory access of an instruction	Kill the offending program
0x10	Data Abort	Bad memory access of data	Kill the offending program
0x14	Reserved	Reserved	Reserved
0x18	Interrupt Request (IRQ)	Hardware wants to make the CPU aware of something	Find out which hardware triggered the interrupt and take appropriate action
0x1C	Fast Interrupt Request (FIQ)	One select hardware can do the above faster than all others	Find out which hardware triggered the interrupt and take appropriate action

Interrupt Requests

An **Interrupt Request** or **IRQ** is a notification to the processor that something happened to that hardware that the processor should know about. This can take many forms, like a keypress or access of privileged memory or receiving a network packet.

In order to determine which hardware devices are allowed to trigger interrupts, and determine which device triggered an interrupt, we need to use the IRQ peripheral, which starts at offset 0xB000 from the peripheral base address. This peripheral has three types of registers: pending, enable, and disable. Pending registers indicate whether a given interrupt has been triggered. These are used in order to determine which hardware device triggered the IRQ exception. Enable registers enable certain interrupts to be triggered by setting the appropriate bit. Disable registers disable certain interrupts by setting the appropriate bit.

The Raspberry Pi has 72 possible IRQs. IRQs 0-63 are shared between the GPU and CPU, and 64-71 are specific to the CPU. The two most important IRQs for our purposes will be the system timer (IRQ number 1) and the USB controller (IRQ number 9).

Here is the layout of the IRQ peripheral:

```
      0       4       8      12      16      20      24      28      32
      +-------------------------------------------------------------+
0x00  |                 ARM Specific Pending (64-71)                |
      +-------------------------------------------------------------+
0x04  |                ARM/GPU Shared Pending 1 (0-31)               |
      +-------------------------------------------------------------+
0x08  |               ARM/GPU Shared Pending 2 (32-63)               |
      +-------------------------------------------------------------+
0x0C  |                       Fast IRQ Control                      |
      +-------------------------------------------------------------+
0x10  |                ARM/GPU Shared Enable 1 (0-31)                |
      +-------------------------------------------------------------+
0x14  |                ARM/GPU Shared Enable 2 (32-63)               |
      +-------------------------------------------------------------+
0x18  |                 ARM Specific Enable (64-72)                 |
      +-------------------------------------------------------------+
0x1C  |                ARM/GPU Shared Disable 1 (0-31)               |
      +-------------------------------------------------------------+
0x20  |                ARM/GPU Shared Disable 2 (32-63)              |
      +-------------------------------------------------------------+
0x24  |                ARM Specific Disable (64-72)                 |
      +-------------------------------------------------------------+
```

There are a couple of unintuitive notes about this peripheral:

1. When you see an interrupt is pending, you should not clear that bit. Each peripheral that can trigger an interrupt has its own mechanism for clearing the pending bit that should be done in the handler for that peripheral's IRQ.
2. Clearing a bit in an enable regiester is neither sufficient nor neccesary to disable an IRQ. An IRQ should only be disabled by setting the correct bit in the disable register

Building an Operating System for the Raspberry Pi

Jake Sandler
jsandler18@gmail.com
jsandler18

Loading the Kernel onto Real Hardware

To load the kernel onto real hardware, you need to do the following:

1. Ensure that you have an SD card that can boot a full operating system like Raspbian
2. Rip out the code from the elf file to form a raw binary file. You can do this by adding `OBJCOPY = ./gcc-arm-none-eabi-X-XXXX-XX-update/bin/arm-none-eabi-objcopy` to the top of the makefile, and adding `$(OBJCOPY) $(IMG_NAME).elf -O binary $(IMG_NAME).img` to the build target.
3. Mount your SD card onto the computer you develop on
4. There should be a file called `kernel.img` if you have a Model 1, and `kernel7.img` for a Model 2 or 3. Rename this to something else.
5. Copy the kernel raw binary to the SD card and name it `kernel.img` on the model 1, and `kernel7.img` on the model 2 and 3.
6. Safely eject the SD card and boot the Raspberry Pi

Building an Operating System for the Raspberry Pi

Jake Sandler
jsandler18@gmail.com

jsandler18

Memory Mapped IO, Peripherals, and Registers

Memory Mapped IO or *MMIO* is the process of interacting with hardware devices by by reading from and writing to predefined memory addresses. All interactions with hardware on the Raspberry Pi occur using MMIO.

A **Peripheral** is a hardware device with a specific address in memory that it writes data to and/or reads data from. All peripherals can be described by an offset from the **Peripheral Base Address**, which starts at 0x20000000 on the Raspberry Pi model 1, and at 0x0x3F000000 on the models 2 and 3.

A **Register** is a 4 byte piece of memory through that a peripheral can read from or write to. These registers are at predefined offsets from the peripheral's base address. For example, it is quite common for one at least one register to be a control register, where each bit in the register corresponds to a certain behavior that the hardware should have. Another common register is a write register, where anything written in it gets sent off to the hardware.

Figuring out where all the peripherals are, what registers they have, and how to use them can mostly be found in the BCM2835 ARM peripheral manual. The BCM2835 is the name of the chipset the Raspberry Pi model 1 uses, and most of the information is good for the model 2 and 3. This document is not easy to parse, and it is missing quite a bit of information, but it is a good starting point, as well as proof that I am not pulling all these addresses out of thin air.

Building an Operating System for the Raspberry Pi

Jake Sandler
jsandler18@gmail.com

jsandler18

The System Timer Peripheral

The **System Timer** is a hardware clock that can be used to keep time and generate interrupts after a certain time. It is located at offset 0x3000 from the peripheral base.

The system timer is a **Free Running Timer** that incriments a 64 bit counter every microsecond, starting as soon as the Pi boots up, and runs in the background for as long as the Pi is on.

There are four **Compare Registers** that the timer compares the low 32 bits low the counter every tick. If any compare register matches the counter, an IRQ is triggered. Each compare register has its own interrupt, interrupt numbers 0-3. Compare registers 0 and 2 are used by the GPU and probably shouldn't be messed with, but 1 and 3 are availible for our use.

There is also a control register. The low four bits are flags that indicate whether or not an interrupt was triggered. Clearing this bit clears the interrupt pending flag for that timer.

Here is the layout of the timer peripheral:

```
      0       4       8       12      16      20      24      28      32
      +-------------------------------------------------------------+
0x00  |0|1|2|3|                       Reserved                      |
      +-------------------------------------------------------------+
0x04  |                         Counter Low 32                      |
      +-------------------------------------------------------------+
0x08  |                         Counter High 32                     |
      +-------------------------------------------------------------+
0x0C  |                        Compare Register 0                   |
      +-------------------------------------------------------------+
0x10  |                        Compare Register 1                   |
      +-------------------------------------------------------------+
0x14  |                        Compare Register 2                   |
      +-------------------------------------------------------------+
0x18  |                        Compare Register 3                   |
      +-------------------------------------------------------------+
```


Jake Sandler
jsandler18@gmail.com
jsandler18