Introduction
============

This file documents the 'goldfish' virtual hardware platform used to run some emulated Android systems under QEMU. It serves as a reference for implementers of virtual devices in QEMU, as well as Linux kernel developers who need to maintain the corresponding drivers.

The following abbreviations will be used here:

```
  $QEMU  -> path to the Android AOSP directory, i.e. a git clone of
            https://android.googlesource.com/platform/external/qemu.git

  $KERNEL -> path to the Android goldfish kernel source tree, i.e. a git clone of
             https://android.googlesource.com/kernel/goldfish.git

             More specifically, to the android-goldfish-2.6.29 branch for now.
```

'goldfish' is the name of a family of similar virtual hardware platforms, that mostly differ in the virtual CPU they support. 'goldfish' started as an ARM-specific platform, but has now been ported to x86 and MIPS virtual CPUs.

Inside of QEMU, goldfish-specific virtual device implementation sources files are in $QEMU/hw/android/goldfish/*.c sources

Inside the Linux kernel tree, they are under $KERNEL/arch/$ARCH/mach-goldfish, or $KERNEL/arch/$ARCH/goldfish/, as well as a few arch-independent drivers in different locations (detailed below).

Goldfish devices appear to the Linux kernel as 'platform devices'. Read [1] and [2] for an introduction and reference documentation for these.

Each device is identified by a name, and an optional unique id (an integer used to distinguish between several identical instances of similar devices, like serial ports, of block devices). When only one instance of a given device can be used, an ID of -1 is used instead.

It also communicates with the kernel through:

- One or more 32-bit of I/O registers, mapped to physical addresses at specific locations which depend on the architecture.

- Zero or more interrupt requests, used to signal to the kernel that an important event occured.

  Note that IRQ lines are numbered from 0 to 31, and are relative to the

```
    goldfish interrupt controller, documented below.


[1] http://lwn.net/Articles/448499/
[2] https://www.kernel.org/doc/Documentation/driver-model/platform.txt


I. Goldfish platform bus:
=========================

The 'platform bus', in Linux kernel speak, is a special device that is capable
of enumerating other platform devices found on the system to the kernel. This
flexibility allows to customize which virtual devices are available when running
a given emulated system configuration.

Relevant files:
  $QEMU/hw/android/goldfish/device.c
  $KERNEL/arch/arm/mach-goldfish/pdev_bus.c
  $KERNEL/arch/x86/mach-goldfish/pdev_bus.c
  $KERNEL/arch/mips/goldfish/pdev_bus.c

Device properties:
  Name: goldfish_device_bus
  Id:   -1
  IrqCount: 1

  32-bit I/O registers (offset, name, abstract)

    0x00 BUS_OP      R: Iterate to next device in enumeration.
                     W: Start device enumeration.

    0x04 GET_NAME    W: Copy device name to kernel memory.
    0x08 NAME_LEN    R: Read length of current device's name.
    0x0c ID          R: Read id of current device.
    0x10 IO_BASE     R: Read I/O base address of current device.
    0x14 IO_SIZE     R: Read I/O base size of current device.
    0x18 IRQ_BASE    R: Read base IRQ of current device.
    0x1c IRQ_COUNT   R: Read IRQ count of current device.

    # For 64-bit guest architectures only:
    0x20 NAME_ADDR_HIGH  W: Write high 32-bit of kernel address of name
                            buffer used by GET_NAME. Must be written to
                            before the GET_NAME write.

The kernel iterates over the list of current devices with something like:

   IO_WRITE(BUS_OP, 0);    // Start iteration, any value other than 0 is invalid.
   for (;;) {
     int ret = IO_READ(BUS_OP);
     if (ret == 0 /* OP_DONE */) {
       // no more devices.
       break;
     }
     else if (ret == 8 /* OP_ADD_DEV */) {
       // Read device properties.
```

```
        Device dev;
        dev.name_len  = IO_READ(NAME_LEN);
        dev.id        = IO_READ(ID);
        dev.io_base   = IO_READ(IO_BASE);
        dev.io_size   = IO_READ(IO_SIZE);
        dev.irq_base  = IO_READ(IRQ_BASE);
        dev.irq_count = IO_READ(IRQ_COUNT);

        dev.name = kalloc(dev.name_len + 1);  // allocate room for device name.
     #if 64BIT_GUEST_CPU
        IO_WRITE(NAME_ADDR_HIGH, (uint32_t)(dev.name >> 32));
     #endif
        IO_WRITE(GET_NAME, (uint32_t)dev.name);  // copy to kernel memory.
        dev.name[dev.name_len] = 0;

        .. add device to kernel's list.
      }
      else {
        // Not returned by current goldfish implementation.
      }
    }

The device also uses a single IRQ, which it will raise to indicate to the kernel
that new devices are available, or that some of them have been removed. The
kernel will then start a new enumeration. The IRQ is lowered by the device only
when a IO_READ(BUS_OP) returns 0 (OP_DONE).

NOTE: The kernel hard-codes a platform_device definition with the name
      "goldfish_pdev_bus" for the platform bus (e.g. see
      $KERNEL/arch/arm/mach-goldfish/board-goldfish.c), however, the bus itself
      will appear during enumeration as a device named "goldfish_device_bus"

      The kernel driver for the platform bus only matches the "goldfish_pdev_bus"
      name, and will ignore any device named "goldfish_device_bus".


II. Goldfish interrupt controller:
==================================

IMPORTANT: The controller IS NOT USED IN EMULATED X86 SYSTEMS.
           TODO(digit): Indicate which virtual PIC is used on x86 systems.

Relevant files:
  $QEMU/hw/android/goldfish/interrupt.c
  $KERNEL/arch/arm/mach-goldfish/board-goldfish.c
  $KERNEL/arch/mips/goldfish/goldfish-interrupt.c

Device properties:
  Name: goldfish_interrupt_controller
  Id: -1
  IrqCount: 0  (uses parent CPU IRQ instead).

  32-bit I/O registers (offset, name, abtract):
    0x00 STATUS       R: Read the number of pending interrupts (0 to 32).
    0x04 NUMBER       R: Read the lowest pending interrupt index, or 0 if none.
```

```
    0x08 DISABLE_ALL  W: Clear all pending interrupts (does not disable them!)
    0x0c DISABLE      W: Disable a given interrupt, value must be in [0..31].
    0x10 ENABLE       W: Enable a given interrupt, value must be in [0..31].
```

Goldfish provides its own interrupt controller that can manage up to 32 distinct maskable interrupt request lines. The controller itself is cascaded from a parent CPU IRQ.

What this means in practice:

- Each IRQ has a 'level' that is either 'high' (1) or 'low' (0).

- Each IRQ also has a binary 'enable' flag.

- Whenever (level == 1 && enabled == 1) is reached due to a state change, the controller raises its parent IRQ. This typically interrupts the CPU and forces the kernel to deal with the interrupt request.

- Raised/Enabled interrupts that have not been serviced yet are called "pending". Raised/Disabled interrupts are called "masked" and are essentially silent until enabled.

When the interrupt controller triggers the parent IRQ, the kernel should do the following:

```
  num_pending = IO_READ(STATUS);  // Read number of pending interrupts.
  for (int n = 0; n < num_pending; ++n) {
    int irq_index = IO_READ(NUMBER);  // Read n-th interrupt index.
    .. service interrupt request with the proper driver.
  }
```

IO_WRITE(DISABLE, <num>) or IO_WRITE(ENABLE, <num>) can change the 'enable' flag of a given IRQ. <num> must be a value in the [0..31] range. Note that enabling an IRQ which has already been raised will make it active, i.e. it will raise the parent IRQ.

IO_WRITE(DISABLE_ALL, 0) can be used to lower all interrupt levels at once (even disabled one). Note that this constant is probably mis-named since it does not change the 'enable' flag of any IRQ.

Note that this is the only way for the kernel to lower an IRQ level through this device. Generally speaking, Goldfish devices are responsible for lowering their own IRQ, which is performed either when a specific condition is met, or when the kernel reads from or writes to a device-specific I/O register.

III. Godlfish timer:

NOTE: This is not used on x86 emulated platforms.

Relevant files:
  $QEMU/hw/android/goldfish/timer.c
  $KERNEL/arch/arm/mach-goldfish/timer.c
  $KERNEL/arch/mips/goldfish/goldfish-time.c

```

Device properties:
  Name: goldfish_timer
  Id: -1
  IrqCount: 1

  32-bit I/O registers (offset, name, abstract)
    0x00  TIME_LOW         R: Get current time, then return low-order 32-bits.
    0x04  TIME_HIGH        R: Return high 32-bits from previous TIME_LOW read.
    0x08  ALARM_LOW        W: Set low 32-bit value of alarm, then arm it.
    0x0c  ALARM_HIGH       W: Set high 32-bit value of alarm.
    0x10  CLEAR_INTERRUPT  W: Lower device's irq level.
    0x14  CLEAR_ALARM

This device is used to return the current host time to the kernel, as a
high-precision signed 64-bit nanoseconds value, starting from a liberal point
in time. This value should correspond to the QEMU "vm_clock", i.e. it should
not be updated when the emulated system does _not_ run, and hence cannot be
based directly on a host clock.

To read the value, the kernel must perform an IO_READ(TIME_LOW), which returns
an unsigned 32-bit value, before an IO_READ(TIME_HIGH), which returns a signed
32-bit value, corresponding to the higher half of the full value.

The device can also be used to program an alarm, with something like:

  IO_WRITE(ALARM_HIGH, <high-value>)  // Must happen first.
  IO_WRITE(ALARM_LOW, <low-value>)    // Must happen second.

When the corresponding value is reached, the device will raise its IRQ. Note
that the IRQ is raised as soon as the second IO_WRITE() if the alarm value is
already older than the current time.

IO_WRITE(CLEAR_INTERRUPT, <any>) can be used to lower the IRQ level once the
alarm has been handled by the kernel.

IO_WRITE(CLEAR_ALARM, <any>) can be used to disarm an existing alarm, if any.

Note: At the moment, the alarm is only used on ARM-based system. MIPS based
      systems only use TIME_LOW / TIME_HIGH on this device.


III. Goldfish real-time clock (RTC):
====================================

Relevant files:
  $QEMU/hw/android/goldfish/timer.c
  $KERNEL/drivers/rtc/rtc-goldfish.c

Device properties:
  Name: goldfish_rtc
  Id: -1
  IrqCount: 1
  I/O Registers:
    0x00  TIME_LOW         R: Get current time, then return low-order 32-bits.
```

```
    0x04  TIME_HIGH        R: Return high 32-bits, from previous TIME_LOW read.
    0x08  ALARM_LOW        W: Set low 32-bit value or alarm, then arm it.
    0x0c  ALARM_HIGH       W: Set high 32-bit value of alarm.
    0x10  CLEAR_INTERRUPT  W: Lower device's irq level.

This device is _very_ similar to the Goldfish timer one, with the following
important differences:

  - Values reported are still 64-bit nanoseconds, but they have a granularity
    of 1 second, and represent host-specific values (really 'time() * 1e9')

  - The alarm is non-functioning, i.e. writing to ALARM_LOW / ALARM_HIGH will
    work, but will never arm any alarm.

To support old Goldfish kernels, make sure to support writing to
ALARM_LOW / ALARM_HIGH / CLEAR_INTERRUPT, even if the device never raises its
IRQ.


IV. Goldfish serial port (tty):
===============================

Relevant files:
  $QEMU/hw/android/goldfish/tty.c
  $KERNEL/drivers/char/goldfish_tty.c
  $KERNEL/arch/arm/mach-goldfish/include/debug-macro.S

Device properties:
  Name: goldfish_tty
  Id: 0 to N
  IrqCount:
  I/O Registers:
    0x00  PUT_CHAR      W: Write a single 8-bit value to the serial port.
    0x04  BYTES_READY   R: Read the number of available buffered input bytes.
    0x08  CMD           W: Send command (see below).
    0x10  DATA_PTR      W: Write kernel buffer address.
    0x14  DATA_LEN      W: Write kernel buffer size.

    # For 64-bit guest CPUs only:
    0x18  DATA_PTR_HIGH    W: Write high 32 bits of kernel buffer address.

This is the first case of a multi-instance goldfish device in this document.
Each instance implements a virtual serial port that contains a small internal
buffer where incoming data is stored until the kernel fetches it.

The CMD I/O register is used to send various commands to the device, identified
by the following values:

  0x00  CMD_INT_DISABLE   Disable device.
  0x01  CMD_INT_ENABLE    Enable device.
  0x02  CMD_WRITE_BUFFER  Write buffer from kernel to device.
  0x03  CMD_READ_BUFFER   Read buffer from device to kernel.

Each device instance uses one IRQ that is raised to indicate that there is
incoming/buffered data to read. To read such data, the kernel should do the
```

following:

```
    len = IO_READ(PUT_CHAR);   // Read length of incoming data.
    if (len == 0) return;      // Nothing to do.

    available = get_buffer(len, &buffer);  // Get address of buffer and its size.
    #if 64BIT_GUEST_CPU
    IO_WRITE(DATA_PTR_HIGH, buffer >> 32);
    #endif
    IO_WRITE(DATA_PTR, buffer);            // Write buffer address to device.
    IO_WRITE(DATA_LEN, available);         // Write buffer length to device.
    IO_WRITE(CMD, CMD_READ_BUFFER);        // Read the data into kernel buffer.
```

The device will automatically lower its IRQ when there is no more input data in its buffer. However, the kernel can also temporarily disable device interrupts with CMD_INT_DISABLE / CMD_INT_ENABLE.

Note that disabling interrupts does not flush the buffer, nor prevent it from buffering further data from external inputs.

To write to the serial port, the device can either send a single byte at a time with:

```
  IO_WRITE(PUT_CHAR, <value>)    // Send the lower 8 bits of <value>.
```

Or use the mode efficient sequence:

```
  #if 64BIT_GUEST_CPU
  IO_WRITE(DATA_PTR_HIGH, buffer >> 32)
  #endif
  IO_WRITE(DATA_PTR, buffer)
  IO_WRITE(DATA_LEN, buffer_len)
  IO_WRITE(CMD, CMD_WRITE_BUFFER)
```

The former is less efficient but simpler, and is typically used by the kernel to send debug messages only.

Note that the Android emulator always reserves the first two virtual serial ports:

- The first one is used to receive kernel messages, this is done by adding the 'console=ttyS0' parameter to the kernel command line in $QEMU/vl-android.c

- The second one is used to setup the legacy "qemud" channel, used on older Android platform revisions. This is done by adding 'android.qemud=ttyS1' on the kernel command line in $QEMU/vl-android.c

  Read docs/ANDROID-QEMUD.TXT for more details about the data that passes through this serial port. In a nutshell, this is required to emulate older Android releases (e.g. cupcake). It provides a direct communication channel between the guest system and the emulator.

  More recent Android platforms do not use QEMUD anymore, but instead rely on the much faster "QEMU pipe" device, described later in this document as

```
  well as in docs/ANDROID-QEMU-PIPE.TXT.


V. Goldfish framebuffer:
========================

Relevant files:
  $QEMU/hw/android/goldfish/fb.c
  $KERNEL/drivers/video/goldfish_fb.c

Device properties:
  Name: goldfish_fb
  Id: 0 to N  (only one used in practice).
  IrqCount: 0
  I/O Registers:
    0x00  GET_WIDTH       R: Read framebuffer width in pixels.
    0x04  GET_HEIGHT      R: Read framebuffer height in pixels.
    0x08  INT_STATUS
    0x0c  INT_ENABLE
    0x10  SET_BASE
    0x14  SET_ROTATION
    0x18  SET_BLANK       W: Set 'blank' flag.
    0x1c  GET_PHYS_WIDTH  R: Read framebuffer width in millimeters.
    0x20  GET_PHYS_HEIGHT R: Read framebuffer height in millimeters.
    0x24  GET_FORMAT      R: Read framebuffer pixel format.

The framebuffer device is a bit peculiar, because it uses, in addition to the
typical I/O registers and IRQs, a large area of physical memory, allocated by
the kernel, but visible to the emulator, to store a large pixel buffer.

The emulator is responsible for displaying the framebuffer content in its UI
window, which can be rotated, as instructed by the kernel.

IMPORTANT NOTE: When GPU emulation is enabled, the framebuffer will typically
only be used during boot. Note that GPU emulation doesn't rely on a specific
virtual GPU device, however, it uses the "QEMU Pipe" device described below.
For more information, please read:

  external/qemu/distrib/android-emugl/DESIGN

On boot, the kernel will read various properties of the framebuffer:

  IO_READ(GET_WIDTH) and IO_READ(GET_HEIGHT) return the width and height of
  the framebuffer in pixels. Note that a 'row' corresponds to consecutive bytes
  in memory, but doesn't necessarily to an horizontal line on the final display,
  due to possible rotation (see SET_ROTATION below).

  IO_READ(GET_PHYS_WIDTH) and IO_READ(GET_PHYS_HEIGHT) return the emulated
  physical width and height in millimeters, this is later used by the kernel
  and the platform to determine the device's emulated density.

  IO_READ(GET_FORMAT) returns a value matching the format of pixels in the
  framebuffer. Note that these values are specified by the Android hardware
  abstraction layer (HAL) and cannot change:

```

```
      0x01  HAL_PIXEL_FORMAT_BRGA_8888
      0x02  HAL_PIXEL_FORMAT_RGBX_8888
      0x03  HAL_PIXEL_FORMAT_RGB_888
      0x04  HAL_PIXEL_FORMAT_RGB_565
      0x05  HAL_PIXEL_FORMAT_BGRA_8888
      0x06  HAL_PIXEL_FORMAT_RGBA_5551
      0x08  HAL_PIXEL_FORMAT_RGBA_4444

    HOWEVER, the kernel driver only expects a value of HAL_PIXEL_FORMAT_RGB_565
    at the moment. Until this is fixed, the virtual device should always return
    the value 0x04 here. Rows are not padded, so the size in bytes of a single
    framebuffer will always be exactly 'width * heigth * 2'.

    Note that GPU emulation doesn't have this limitation and can use and display
    32-bit surfaces properly, because it doesn't use the framebuffer.

  The device has a 'blank' flag. When set to 1, the UI should only display an
  empty/blank framebuffer, ignoring the content of the framebuffer memory.
  It is set with IO_WRITE(SET_BLANK, <value>), where value can be 1 or 0. This is
  used when simulating suspend/resume.

  IMPORTANT: The framebuffer memory is allocated by the kernel, which will send
  its physical address to the device by using IO_WRITE(SET_BASE, <address>).

  The kernel really allocates a memory buffer large enough to hold *two*
  framebuffers, in order to implement panning / double-buffering. This also means
  that calls to IO_WRITE(SET_BASE, <address>) will be frequent.

  The allocation happens with dma_alloc_writecombine() on ARM, which can only
  allocate a maximum of 4 MB, this limits the size of each framebuffer to 2 MB,
  which may not be enough to emulate high-density devices :-(

  For other architectures, dma_alloc_coherent() is used instead, and has the same
  upper limit / limitation.

  TODO(digit): Explain how it's possible to raise this limit by modifyinf
               CONSISTENT_DMA_SIZE and/or MAX_ORDER in the kernel configuration.

  The device uses a single IRQ to notify the kernel of several events. When it
  is raised, the kernel IRQ handler must IO_READ(INT_STATUS), which will return
  a value containing the following bit flags:

    bit 0: Set to 1 to indicate a VSYNC event.

    bit 1: Set to 1 to indicate that the content of a previous SET_BASE has
           been properly displayed.

  Note that reading this register also lowers the device's IRQ level.

  The second flag is essentially a way to notify the kernel that an
  IO_WRITE(SET_BASE, <address>) operation has been succesfully processed by
  the emulator, i.e. that the new content has been displayed to the user.

  The kernel can control which flags should raise an IRQ by using
  IO_WRITE(INT_ENABLE, <flags>), where <flags> has the same format as the
```

```
result of IO_READ(INT_STATUS). If the corresponding bit is 0, the an IRQ
for the corresponding event will never be generated,


VI. Goldfish audio device:
==========================

Relevant files:
  $QEMU/hw/android/goldfish/audio.c
  $KERNEL/drivers/misc/goldfish_audio.c

Device properties:
  Name: goldfish_audio
  Id: -1
  IrqCount: 1
  I/O Registers:
    0x00  INT_STATUS
    0x04  INT_ENABLE
    0x08  SET_WRITE_BUFFER_1     W: Set address of first kernel output buffer.
    0x0c  SET_WRITE_BUFFER_2     W: Set address of second kernel output buffer.
    0x10  WRITE_BUFFER_1         W: Send first kernel buffer samples to output.
    0x14  WRITE_BUFFER_2         W: Send second kernel buffer samples to output.
    0x18  READ_SUPPORTED         R: Reads 1 if input is supported, 0 otherwise.
    0x1c  SET_READ_BUFFER
    0x20  START_READ
    0x24  READ_BUFFER_AVAILABLE

    # For 64-bit guest CPUs
    0x28  SET_WRITE_BUFFER_1_HIGH  W: Set high 32 bits of 1st kernel output buffer address.
    0x30  SET_WRITE_BUFFER_2_HIGH  W: Set high 32 bits  of 2nd kernel output buffer address.
    0x34  SET_READ_BUFFER_HIGH     W: Set high 32 bits of kernel input buffer address.

This device implements a virtual sound card with the following properties:

  - Stereo output at fixed 44.1 kHz frequency, using signed 16-bit samples.
    Mandatory.

  - Mono input at fixed 8 kHz frequency, using signed 16-bit samples.
    Optional.

For output, the kernel driver allocates two internal buffers to hold output
samples, and passes their physical address to the emulator as follows:

  #if 64BIT_GUEST_CPU
  IO_WRITE(SET_WRITE_BUFFER_1_HIGH, (uint32_t)(buffer1 >> 32));
  IO_WRITE(SET_WRITE_BUFFER_2_HIGH, (uint32_t)(buffer2 >> 32));
  #endif
  IO_WRITE(SET_WRITE_BUFFER_1, (uint32_t)buffer1);
  IO_WRITE(SET_WRITE_BUFFER_2, (uint32_t)buffer2);

After this, samples will be sent from the driver to the virtual device by
using one of IO_WRITE(WRITE_BUFFER_1, <length1>) or
IO_WRITE(WRITE_BUFFER_2, <length2>), depending on which sample buffer to use.
NOTE: Each length is in bytes.
```

Note however that the driver should wait, before doing this, until the device gives permission by raising its IRQ and setting the appropriate 'status' flags.

The virtual device has an internal 'int_status' field made of 3 bit flags:

```
  bit0: 1 iff the device is ready to receive data from the first buffer.
  bit1: 1 iff the device is ready to receive data from the second buffer.
  bit2: 1 iff the device has input samples for the kernel to read.
```

Note that an IO_READ(INT_STATUS) also automatically lowers the IRQ level, except if the read value is 0 (which should not happen, since it should not raise the IRQ).

The corresponding interrupts can be masked by using IO_WRITE(INT_ENABLE, <mask>), where <mask> has the same format as 'int_status'. A 1 bit in the mask enables the IRQ raise when the corresponding status bit is also set to 1.

For input, the driver should first IO_READ(READ_SUPPORTED), which will return 1 if the virtual device supports input, or 0 otherwise. If it does support it, the driver must allocate an internal buffer and send its physical address with IO_WRITE(SET_READ_BUFFER, <read-buffer>) (with a previous write to SET_READ_BUFFER_HIGH on 64-bit guest CPUS), then perform IO_WRITE(START_READ, <read-buffer-length>) to start recording and specify the kernel's buffer length.

Later, the device will raise its IRQ and set bit2 of 'int_status' to indicate there are incoming samples to the driver. In its interrupt handler, the latter should IO_READ(READ_BUFFER_AVAILABLE), which triggers the transfer (from the device to the kernel), as well as return the size in bytes of the samples.

VII. Goldfish battery:
======================

```
Relevant files:
  $QEMU/hw/android/goldfish/battery.c
  $QEMU/hw/power_supply.h
  $KERNEL/drivers/power/goldfish_battery.c

Device properties:
  Name: goldfish_battery
  Id: -1
  IrqCount: 1
  I/O Registers:
    0x00 INT_STATUS   R: Read battery and A/C status change bits.
    0x04 INT_ENABLE   W: Enable or disable IRQ on status change.
    0x08 AC_ONLINE    R: Read 0 if AC power disconnected, 1 otherwise.
    0x0c STATUS       R: Read battery status (charging/full/... see below).
    0x10 HEALTH       R: Read battery health (good/overheat/... see below).
    0x14 PRESENT      R: Read 1 if battery is present, 0 otherwise.
    0x18 CAPACITY     R: Read battery charge percentage in [0..100] range.
```

A simple device used to report the state of the virtual device's battery, and whether the device is powered through a USB or A/C adapter.

```
The device uses a single IRQ to notify the kernel that the battery or A/C status
changed. When this happens, the kernel should perform an IO_READ(INT_STATUS)
which returns a 2-bit value containing flags:

  bit 0: Set to 1 to indicate a change in battery status.
  bit 1: Set to 1 to indicate a change in A/C status.

Note that reading this register also lowers the IRQ level.

The A/C status can be read with IO_READ(AC_ONLINE), which returns 1 if the
device is powered, or 0 otherwise.

The battery status is spread over multiple I/O registers:

  IO_READ(PRESENT) returns 1 if the battery is present in the virtual device,
  or 0 otherwise.

  IO_READ(CAPACITY) returns the battery's charge percentage, as an integer
  between 0 and 100, inclusive. NOTE: This register is probably misnamed since
  it does not represent the battery's capacity, but it's current charge level.

  IO_READ(STATUS) returns one of the following values:

    0x00  UNKNOWN      Battery state is unknown.
    0x01  CHARGING     Battery is charging.
    0x02  DISCHARGING  Battery is discharging.
    0x03  NOT_CHARGING Battery is not charging (e.g. full or dead).

  IO_READ(HEALTH) returns one of the following values:

    0x00  UNKNOWN         Battery health unknown.
    0x01  GOOD            Battery is in good condition.
    0x02  OVERHEATING     Battery is over-heating.
    0x03  DEAD            Battery is dead.
    0x04  OVERVOLTAGE     Battery generates too much voltage.
    0x05  UNSPEC_FAILURE  Battery has unspecified failure.

The kernel can use IO_WRITE(INT_ENABLE, <flags>) to select which condition
changes should trigger an IRQ. <flags> is a 2-bit value using the same format
as INT_STATUS.


VIII. Goldfish events device (user input):
==========================================

Relevant files:
  $QEMU/hw/android/goldfish/events_device.c
  $KERNEL/drivers/input/keyboard/goldfish_events.c

Device properties:
  Name: goldfish_events
  Id: -1
  IrqCount: 1
  I/O Registers:
    0x00 READ       R: Read next event type, code or value.
```

```
    0x00 SET_PAGE   W: Set page index.
    0x04 LEN        R: Read length of page data.
    0x08 DATA       R: Read page data.
    ....            R: Read additional page data (see below).

This device is responsible for sending several kinds of user input events to
the kernel, i.e. emulated device buttons, hardware keyboard, touch screen,
trackball and lid events.

NOTE: Android supports other input devices like mice or game controllers
      through USB or Bluetooth, these are not supported by this virtual
      Goldfish device.

NOTE: The 'lid event' is useful for devices with a clamshell of foldable
      keyboard design, and is used to report when it is opened or closed.

As per Linux conventions, each 'emulated event' is sent to the kernel as a
series of (<type>,<code>,<value>) triplets or 32-bit values. For more
information, see:

  https://www.kernel.org/doc/Documentation/input/input.txt

As well as the <linux/input.h> kernel header.

Note that in the context of goldfish:

 - Button and keyboard events are reported with:
     (EV_KEY, <code>, <press>)

   Where <code> is a 9-bit keycode, as defined by <linux/input.h>, and
   <press> is 1 for key/button presses, and 0 for releases.

 - For touchscreen events, a single-touch event is reported with:
     (EV_ABS, ABS_X, <x-position>) +
     (EV_ABS, ABS_Y, <y-position>) +
     (EV_ABS, ABS_Z, 0) +
     (EV_KEY, BTN_TOUCH, <button-state>) +
     (EV_SYN, 0, 0)

   where <x-position> and <y-position> are the horizontal and vertical position
   of the touch event, respectfully, and <button-state> is either 1 or 0 and
   indicates the start/end of the touch gesture, respectively.

 - For multi-touch events, things are much more complicated. In a nutshell,
   these events are reported through (EV_ABS, ABS_MT_XXXXX, YYY) triplets,
   as documented at:

   https://www.kernel.org/doc/Documentation/input/multi-touch-protocol.txt

   TODO(digit): There may be bugs in either the virtual device or driver code
                when it comes to multi-touch. Iron out the situation and better
                explain what's required to support all Android platforms.

 - For trackball events:
     (EV_REL, REL_X, <x-delta>) +
```

```
      (EV_REL, REL_Y, <y-delta>) +
      (EV_SYN, 0, 0)

    Where <x-delta> and <y-delta> are the signed relative trackball displacement
    in the horizontal and vertical directions, respectively.

  - For lid events:
      (EV_SW, 0, 1) + (EV_SYN, 0, 0)    // When lid is closed.
      (EV_SW, 0, 0) + (EV_SYN, 0, 0)    // When lid is opened.

When the kernel driver starts, it will probe the device to know what kind
of events are supported by the emulated configuration. There are several
categories of queries:

  - Asking for the current physical keyboard 'charmap' name, used by the system
    to translate keycodes in actual characters. In practice, this will nearly
    always be 'goldfish' for emulated systems, but this out of spec for this
    document.

  - Asking which event codes are supported for a given event type
    (e.g. all the possible KEY_XXX values generated for EV_KEY typed triplets).

  - Asking for various minimum or maximum values for each supported EV_ABS
    event code. For example the min/max values of (EV_ABS, ABS_X, ...) triplets,
    to know the bounds of the input touch panel.

The kernel driver first select which kind of query it wants by using
IO_WRITE(SET_PAGE, <page>), where <page> is one of the following values:

    PAGE_NAME    0x0000   Keyboard charmap name.
    PAGE_EVBITS  0x10000  Event code supported sets.
    PAGE_ABSDATA 0x20003  (really 0x20000 + EV_ABS) EV_ABS min/max values.

Once a 'page' has been selected, it is possible to read from it with
IO_READ(LEN) and IO_READ(DATA). In practice:

  - To read the name of the keyboard charmap, the kernel will do:

      IO_WRITE(SET_PAGE, PAGE_NAME);  # Ind

      charmap_name_len = IO_READ(LEN);
      charmap_name = kalloc(charmap_name_len + 1);
      for (int n = 0; n < charmap_name_len; ++n)
        charmap_name[n] = (char) IO_READ(DATA);
      charmap_name[n] = 0;

  - To read which codes a given event type (here EV_KEY) supports:

      IO_WRITE(SET_PAGE, PAGE_EVBITS + EV_KEY);  // Or EV_REL, EV_ABS, etc...

      bitmask_len = IO_READ(LEN);
      for (int offset = 0; offset < bitmask_len; ++offset) {
          uint8_t mask = (uint8_t) IO_READ(DATA):
          for (int bit = 0; bit < 8; ++bit) {
              int code = (offset * 8) + bit;
```

```
              if ((mask & (1 << bit)) != 0) {
                  ... record that keycode |code| is supported.
              }
          }
      }
```

  - To read the range values of absolute event values:

```
      IO_WRITE(SET_PAGE, PAGE_ABSDATA);
      max_entries = IO_READ(LEN);
      for (int n = 0; n < max_entries; n += 4) {
        int32_t min = IO_READ(DATA + n);
        int32_t max = IO_READ(DATA + n + 4);
        int32_t fuzz = IO_READ(DATA + n + 8);
        int32_t flat = IO_READ(DATA + n + 12);
        int event_code = n/4;

        // Record (min, max, fuzz, flat) values for EV_ABS 'event_code'.
      }
```

    Note that the 'fuzz' and 'flat' values reported by Goldfish are always 0, refer to the source for more details.

At runtime, the device implements a small buffer for incoming event triplets (each one is stored as three 32-bit integers in a circular buffer), and raises its IRQ to signal them to the kernel.

When that happens, the kernel driver should use IO_READ(READ) to extract the 32-bit values from the device. Note that three IO_READ() calls are required to extract a single event triplet.

There are a few important notes here:

  - The IRQ should not be raised _before_ the kernel driver is started (otherwise the driver will be confused and ignore all events).

    I.e. the emulator can buffer events before kernel initialization completes, but should only raise the IRQ, if needed, lazily. Currently this is done on the first IO_READ(LEN) following a IO_WRITE(SET_PAGE, PAGE_ABSDATA).

  - The IRQ is lowered by the device once all event values have been read, i.e. its buffer is empty.

    However, on x86, if after an IO_READ(READ), there are still values in the device's buffer, the IRQ should be lowered then re-raised immediately.

IX. Goldfish NAND device:

Relevant files:
  $QEMU/hw/android/goldfish/nand.c
  $KERNEL/drivers/mtd/devices/goldfish_nand.c

Device properties:

```
  Name: goldfish_nand
  Id: -1
  IrqCount: 1
  I/O Registers:

This virtual device can provide access to one or more emulated NAND memory
banks [3] (each one being backed by a different host file in the current
implementation).

These are used to back the following virtual partition files:

  - system.img
  - data.img
  - cache.img

TODO(digit): Complete this.


[3] http://en.wikipedia.org/wiki/Flash_memory#NAND_memories


X. Goldfish MMC device:
=======================

Relevant files:
  $QEMU/hw/android/goldfish/mmc.c
  $KERNEL/drivers/mmc/host/goldfish.c

Device properties:
  Name: goldfish_mmc
  Id: -1
  IrqCount: 1
  I/O Registers:

Similar to the NAND device, but uses a different, higher-level interface
to access the emulated 'flash' memory. This is only used to access the
virtual SDCard device with the Android emulator.

TODO(digit): Complete this.


XIV. QEMU Pipe device:
======================

Relevant files:
  $QEMU/hw/android/goldfish/pipe.c
  $KERNEL/drivers/misc/qemupipe/qemu_pipe.c

Device properties:
  Name: qemu_pipe
  Id: -1
  IrqCount: 1
  I/O Registers:
    0x00  COMMAND          W: Write to perform command (see below).
    0x04  STATUS           R: Read status
```

```
    0x08  CHANNEL          RW: Read or set current channel id.
    0x0c  SIZE             RW: Read or set current buffer size.
    0x10  ADDRESS          RW: Read or set current buffer physical address.
    0x14  WAKES            R: Read wake flags.
    0x18  PARAMS_ADDR_LOW  RW: Read/set low bytes of parameters block address.
    0x1c  PARAMS_ADDR_HIGH RW: Read/set high bytes of parameters block address.
    0x20  ACCESS_PARAMS    W: Perform access with parameter block.

This is a special device that is totally specific to QEMU, but allows guest
processes to communicate directly with the emulator with extremely high
performance. This is achieved by avoiding any in-kernel memory copies, relying
on the fact that QEMU can access guest memory at runtime (under proper
conditions controlled by the kernel).

Please refer to $QEMU/docs/ANDROID-QEMU-PIPE.TXT for full details on the
device's operations.


XIII. QEMU Trace device:
========================

Relevant files:
  $QEMU/hw/android/goldfish/trace.c
  $KERNEL/drivers/misc/qemutrace/qemu_trace.c
  $KERNEL/drivers/misc/qemutrace/qemu_trace_sysfs.c
  $KERNEL/fs/exec.c
  $KERNEL/exit.c
  $KERNEL/fork.c
  $KERNEL/sched/core.c
  $KERNEL/mm/mmap.c

Device properties:
  Name: qemu_trace
  Id: -1
  IrqCount: 0
  I/O Registers:

TODO(digit)
```