Symbian ADT Sourcery G++ Lite

ARM SymbianOS

Symbian ADT Sourcery G++ Lite 4.4-172

Symbian Virtual Platform Documentation

Symbian ADT Sourcery G++ Lite: ARM SymbianOS: Symbian ADT Sourcery G++ Lite 4.4-172: Symbian Virtual Platform Documentation

CodeSourcery, Inc.

Abstract

This document describes the Symbian Virtual Platform, including how to configure and customize the virtual environment, and programming details for the base devices.

Table of Contents

Preface

This preface introduces the *Symbian Virtual Platform Documentation*. It explains the structure of this guide and lists other sources of information that relate to Symbian ADT Sourcery G++ Lite

1. Intended Audience

This document is written for developers targeting the Symbian Virtual Platform. It describes how to customize the configuration of the base virtual platform, and the operation of the devices within that system. It also describes how to extend the virtual platform by adding emulation of additional devices.

It is assumed that you are already familiar with the process of bringing up a SymbianOS base port. The tasks required to build a SymbianOS image for the Virtual Platform are not covered.

2. Organisation

This document is organised into the following chapters and appendices:

Chapter 1, “Configuring the Virtual Platform”	This chapter describes how to configure the Symbian Virtual Platform.
Chapter 2, “GUI Configuration”	This chapter describes how to configure and bind devices to the GUI.
Chapter 3, “Base Platform Devices”	This chapter describes the operation of the virtual devices that make up the base virtual platform.
Chapter 4, “Virtio”	This chapter describes the Virtio framework used by some of the virtual IO devices.
Chapter 5, “Device Plugins”	This chapter describes how to extend the virtual platform by adding emulation of new devices.

3. Typographical Conventions

The following typographical conventions are used in this guide:

`> command arg ...`	A command, typed by the user, and its output. The “>” character is the command prompt.
`command`	The name of a program, when used in a sentence, rather than in literal input or output.
`literal`	Text provided to or received from a computer program.
`placeholder`	Text that should be replaced with an appropriate value when typing a command.
`\`	At the end of a line in command or program examples, indicates that a long line of literal input or output continues onto the next line in the document.

Chapter 1
Configuring the Virtual Platform

Ths Symbian Virtual Platform is a flexible system model that can be reconfigured to provide different devices, memory layouts, etc. This chapter describes the configuration files used to achieve this.

1.1. About Flattened Device Trees

A Symbian Virtual Platform board configuration is described using Flattened Device Trees. These are based on the device trees used by OpenFirmware (IEEE 1275-1994) systems. Flattened Device Trees were originally designed as a means of communication between a bootloader and a PowerPC Linux kernel. However despite this they are not PowerPC- or Linux-specific, and hove been adopted as a means of describing a Symbian Virtual Platform board configuration.

Flattened Device Trees can be represented in two different forms. The source form is an ASCII text representation suitable for editing by humans. The binary form is a compact machine-readable representation used by the Virtual Platform. The Device Tree Compiler utility is provided to convert between these two forms.

A Flattened Device Tree consists of a set of nodes. Each node represents a device or bus, and has a set of properties. Typically these properties describe what type of device this node represents, where interrupts are routed, where memory mapped control registers should be placed, and any other device-specific properties.

Each node has a unique name. This name consists of two parts separated by an `@` symbol. The first part is the base name, which is typically the same for different instances of similar devices. The second part is the device address, which is used to distinguish multiple instances of similar devices. The node name has no effect on the operation of the device; it is just used to locate the device within the tree.

Each property consists of a *`name=value`* pair. The value may be either empty, a null-terminated ASCII string, raw byte data, or a set of 32-bit cells (values). Cells are generally used to hold addresses, sizes, references to other devices, or general numeric values. A cell value can be specified as either a reference to a device node, or a hexadecimal number.

1.2. Board Configuration Files

QEMU uses a board configuration file to create the virtual machine. This configuration file should be a binary Flattened Device Tree describing the virtual machine.

1.2.1. Invoking QEMU

Board configuration files can be specified with the `-M` command-line option. For example:

```
> arm-none-symbianelf-qemu-system -M board.dtb
```

1.2.2. Compiling Device Trees

The Device Tree Compiler is a utility that converts human-readable Flattened Device Tree source files into machine-readable binary representation. The following command converts a source file (*`board`*`.dts`) to a binary file (*`board`*`.dtb`).

```
> arm-none-symbianelf-dtc board.dts -O dtb -o board.dtb
```

1.3. Structure of Board Configuration Trees

A typical board contains three sets of device nodes: CPU, Memory and Peripherals.

1.3.1. CPU Device Nodes

The CPU device node should be located in the `/cpus` branch of the device tree. The `/cpus` node should have `#address-cells=1` and `#size-cells=0`. The name of the node is used to determine the type of the CPU. Valid properties are:

`device-type` Must be set to `"cpu"`.

`cp15,ctr` (optional) The value of the CP15 Cache Type Register.

`cp15,clid` (optional) The value of the CP15 Cache Level ID Register.

`cp15,ccsid`*`N`* (optional) The value of the CP15 Cache Size ID Register for data/unified cache level *`N`*.

`cp15,ccsid`*`N`*`i` (optional) The value of the CP15 Cache Size ID Register for instruction cache level *`N`*.

1.3.2. Memory Device Nodes

Memory device nodes describe areas of RAM. They should be located in the root of the device tree. The root of the device tree should have `#address-cells=1` and `#size-cells=1`. Multiple memory regions may be added as separate nodes or combined into a single node. Valid properties are:

`device-type` Must be set to `"memory"`.

`reg` Each pair of cells specifies an {address, length} of a memory region.

1.3.3. Peripheral Device Nodes

This includes all other devices in the system, including interrupt controllers. They should be located in a branch of the device tree with `#address-cells=1` and `#size-cells=0`.

Devices are typically identified by the `compatible` property, and have a set of memory-mapped registers located by the `reg` property. For individual device details see Chapter 3, “Base Platform Devices”.

1.3.4. Interrupts

Interrupt routing is determined by the `interrupt`, `interrupt-parent` and `qemu,interrupts` properties.

In most cases interrupt routing is specified using a subset of the OpenFirmware interrupt trees. The `interrupt-parent` property specifies the parent interrupt device, and the `interrupts` property specifies how device interrupts map onto parent interrupts.

The current implementation requires that interrupt parents have `#interrupt-cells=1`. The `interrupt-map` property is not implemented.

In some cases (typically routing from the top-level interrupt controller to the CPU), an OpenFirmware interrupt tree does not include this link. The `qemu,interrupts` property is used to describe this to QEMU. This property consists of a pair of cells for each device interrupts. Each pair is a parent device reference and interrupt number.

1.4. Example

Below is an example of a simple board description file. It constitutes a Cortex-A8 cpu, ram, interrupt controller and a serial port.

```
/ {
    #address-cells = <1>;
    #size-cells  = <1>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu0: ARM,Cortex-A8@0 {
            device_type = "cpu";
            reg = <0>;
        };
    };
    memory@0 {
        device_type = "memory";
        /* 128Mb ram at address zero.  */
        reg = <0 08000000>;
    };
    syborg {
        #address-cells = <1>;
        #size-cells = <0>;
        /* Interrupt controller at address 0xc0000000.  */
        intc: intc@0 {
            compatible = "syborg,interrupt";
            #interrupt-cells = <1>;
            reg = <c0000000>;
            interrupt-controller;
            qemu,interrupts = <&cpu0 0>;
            num-interrupts = <20>;
        };
        /* Serial port at address 0xc0006000.  */
        serial@0 {
            device_type = "serial";
            compatible = "syborg,serial";
            chardev = "serial0";
            reg = <c0006000>;
            interrupts = <5>;
            interrupt-parent = <&intc>;
        };
    };
};
```

Chapter 2
GUI Configuration

The Symbian Virtual Platform includes a configurable GUI. The GUI provides means to set skins and combine graphic displays, text displays, pointer devices, and graphic widgets in virtual terminals, without needing to recompile the system. This chapter documents the structure of the GUI and its configuration.

2.1. GUI Structure and Components

The GUI configuration is specified in an XML file. The structure of the XML file is the following:

- buttons
- display areas
- pointer areas

The GUI is specified as a collection of Virtual Terminals.

Both frame buffer and pointer devices are referenced by their FDT device name, which is specified in the `devid` attribute.

2.2. Using the GUI

The GUI is enabled with the `-gui` command-line option of QEMU, followed by the XML file name. For example:

```
./arm-none-symbianelf-qemu-system -kernel ./a.out -M syborg.dtb \
-gui syborg.xml
```

When the GUI loads, relative image pathnames in the XML file are looked up relative to the directory in which QEMU is launched, unless they include a path in the file name (either the `background` element or the `pressedimg` element for Virtual Terminals and buttons respectively).

Currently, only the PNG image file format is accepted.

During the GUI operation, the following key combinations are available:

- Ctrl+Alt+1..9: switches between VTs
- Ctrl+Alt+F: toggles fullscreen
- Ctrl+Alt: exits grabbing mode when active

2.3. GUI XML File Specification

The top-level element in the GUI XML file is a `<gui>` element.

2.3.1. Virtual Terminals

Virtual Terminals are backgrounds onto which buttons, display areas and frame buffers can be placed.

The virtual terminals are specified by the `<vt>` .. `</vt>` block.

Inside the `vt` element, buttons, display areas and pointer areas are specified by `<button>`, `<displayarea>`, and `<pointerarea>` elements respectively. The display and pointer areas can overlap.

Table 2.1. `vt` Element Attributes

Attribute	Description	Mandatory
background	File name of the background image.	yes

Attribute	Description	Mandatory
width	Width of the background. Image cropped if a smaller number than the actual image width is specified.	
height	Height of the background. Image cropped if a smaller number than the actual image height is specified.	

2.3.2. Buttons

Buttons are rectangular regions of the background that can be clicked with a mouse of the host system.

Each button has an associated action which is triggered both when the button is left-button clicked and when it is released. Currently, the only action available is `"sendkey"`, which sends a scan code to the registered keyboard device when the button is pressed, and sends the scan code with the proper bits set when released.

Each button is specified by the `<button .. />` empty element (this means that there is no `</button>` closing tag).

Buttons can be associated with an image, which is displayed when the button is pressed. This image is located in the same region where the button can be clicked, or in a different location through the `pressed_x`, `pressed_y`, `pressed_width`, `pressed_height` attributes (where `pressed_width` and `pressed_height` default to the pressed image width and size respectively).

The button element has the following attributes:

Table 2.2. `button` Element Attributes

Attribute	Description	Mandatory
x	Left coordinate of the clickable area.	yes
y	Top coordinate of the clickable area.	yes
width	Width of the clickable area.	
height	Height of the clickable area.	
pressedimg	File name of the image to display when the button is pressed.	
action	Action to perform when the button is pressed. Only `"sendkey"` is currently supported.	yes
parameter	Parameter for the specified action. Only an integer value representing the key scancode is supported. All parameters shall be enclosed in `""` double quotes.	yes
pressed_x	Left coordinate of the pressed button image. Defaults to the left coordinate of the clickable area.	
pressed_y	Top coordinate of the pressed button image. Defaults to the top coordinate of the clickable area.	
pressed_width	Width of the pressed button image. Defaults to the pressed image width.	
pressed_height	Height of the pressed button image. Defaults to the pressed image height.	

2.3.3. Display Areas

A display area is the region of the VT where a frame buffer device is rendered. There can exist many display areas in the same VT.

Each display area is specified by a `<displayarea .. />` empty element (this means that there is no `</displayarea>` closing tag).

Display areas belonging to a skinned VT cannot be resized or rotated (just flipped vertically or horizontally).

Each display area is associated with a device instance of the FDT through the device name (`devid` attribute).

A display area element has the following attributes:

Table 2.3. `displayarea` Element Attributes

Attribute	Description	Mandatory
x	Left coordinate of the display area.	yes
y	Top coordinate of the display area.	yes
width	Width of the display area.	
height	Height of the display area.	
devid	FDT name of the frame buffer device instance.	yes

2.3.4. Pointer Areas

A pointer area is the region of the VT that forwards its pointer events to a pointer device. A pointer area may grab mouse events. There can exist many pointer areas in the same VT.

Each pointer areas is specified by a `<pointerarea .. />` empty element (this means that there is no `</pointerarea>` closing tag).

Each pointer area is associated with a pointer device instance of the FDT through the device name (`devid` attribute).

A pointer area element has the following attributes:

Table 2.4. `pointerarea` Element Attributes

Attribute	Description	Mandatory
x	Left coordinate of the pointer area.	yes
y	Top coordinate of the pointer area.	yes
width	Width of the pointer area.	
height	Height of the pointer area.	
devid	FDT name of the pointer device instance.	yes
grabonclick	Specifies whether the GUI starts grabbing all mouse events when the first left-button click event occurs, or each event is forwarded to the pointer device without grabbing. 0 = No, 1 = Yes.	

2.4. Sample XML File

The following example demonstrates a configuration for a GUI having two virtual terminals, the first with a button, a display area, and a a pointer area, and the second with just a display area.

```
<gui>
 <vt background="syborg.png" width="600" height="480">
  <button x="20" y="206" width="26" height="14" action="sendkey"
   parameter="1" pressedimg="pressed_button.png
   pressedimg_x="21" pressedimg_y="207"/>
  <displayarea devid="framebuffer@0" x="22" y="40"
   width="100" height="105"/>
  <pointerarea devid="touchscreen@0" x="22" y="40"
   width="100" height="105" grabonclick="0"/>
 </vt>
 <vt background="syborg_back.png" width="600" height="480">
  <displayarea devid="0" x="20" y="20" width="70" height="70"/>
  </vt>
</gui>
```

Chapter 3
Base Platform Devices

The Symbian Virtual Platform includes a base set of virtual devices. These devices are designed to provide essential functionality, while remaining simple to program, and avoiding many of the problems encountered when using real devices. This chapter documents the functionality and programming interfaces for these devices.

3.1. Common Device Characteristics

Unless otherwise specified, each device responds to a single 4-Kbyte region of address space, which consists of several control registers. Each register is 4 bytes wide, and should be accessed with an aligned 32-bit access. Registers may be read-only, write-only or read-write. All other accesses to these device regions must be avoided.

The `reg` property consists of a single cell that specifies the base address of this region. The address must be on a 4-Kbyte boundary.

The compatible attribute is used to identify the type of device. The name of the device is ignored, though it should be unique.

3.2. Platform Device

3.2.1. Description

This device provides platform configuration data. It allows dynamic enumeration and configuration of other devices.

Configuration data is presented as a binary Flattened Device Tree (FDT) data structure.

This device occupies a 16-Mbyte window of address space. The first 4 Kbytes of this address space are control registers, as with other devices. The remainder of this region may be accessed as normal RAM.

3.2.2. Device Properties

```
compatible  syborg,platform
```

3.2.3. Registers

Table 3.1. Platform Device Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d1000	ID	Peripheral ID.
0x004	R	impl def	TREE_START	The address of the FDT header. This is an offset from the platform device base address.

3.3. Interrupt Controller

3.3.1. Description

This device is a simple interrupt controller. It has multiple input IRQs and a single output IRQ. Each input IRQ can be enabled and disabled independently. In addition all inputs can be disabled in a single operation.

All inputs are level-triggered. An input is *active* if it is enabled and raised.

The output IRQ is asserted iff there are any active IRQs.

3.3.2. Device Properties

`compatible` `syborg,interrupt`

`num-interrupts` Number of interrupt sources (optional, default 64).

3.3.3. Registers

Table 3.2. Interrupt Controller Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d0000	ID	Peripheral ID.
0x004	R	0x00000000	STATUS	Number of active interrupts.
0x008	R	0xffffffff	CURRENT	Current active interrupt. Lower numbers are higher priority. 0xffffffff if no interrupts are active.
0x00c	W	N/A	DISABLE_ALL	Writing any value disables all inputs.
0x010	W	N/A	DISABLE	Disable a single interrupt.
0x014	W	N/A	ENABLE	Enable a single interrupt.
0x018	R	impl def	TOTAL	The total number of inputs (active or inactive).

3.4. Serial Port

3.4.1. Description

This device implements a DMA-capable serial port. Bytes written to the DATA register are sent immediately. Received bytes are stored in a FIFO. A maskable interrupt is raised when the data FIFO is not empty.

DMA operates independently for transmit and receive, with a simple address/count pair for each. A DMA transfer is initiated by writing the address register, then the count register. The transfer starts automatically when a nonzero value is written to the count register. As the transfer progresses the address register is incremented and the count register decremented. The transfer stops when the count reaches zero. A maskable interrupt is raised when the count register is zero.

A receive DMA transfer first reads any bytes present in the FIFO, then additional incoming bytes.

The count register may be read at any time to monitor progress. To stop a transfer early, write zero to the count register, then read the address register to determine how many bytes were transferred.

> **Note**
>
> The current implementation completes DMA transmits immediately. However future implementations may transfer asynchronously. Drivers should check whether the previous transmit has completed before starting another one.

3.4.2. Device Properties

`compatible` `syborg,serial`

`chardev` Name of associated character device (e.g. `"serial0"`).

`fifo-size` Size in bytes of the receive FIFO (optional, default 16).

3.4.3. Registers

Table 3.3. Serial Controller Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d1001	ID	Peripheral ID.
0x004	RW	0xffffffff	DATA	Write sends a byte of data. The high 24 bits should be zero. Read removes the next byte from the FIFO and zero extends. If the FIFO is empty then 0xffffffff is read.
0x008	R	0x00000000	FIFO_COUNT	The number of bytes currently in the FIFO.
0x00c	RW	0x00000000	INT_ENABLE	Bit flags to determine which interrupts are enabled. A zero bit masks the interrupt. A nonzero bit enables the interrupt. Bit0 FIFO not empty. Bit1 TX DMA count zero. Bit2 RX DMA count zero.
0x010	RW	0x00000000	DMA_TX_ADDR	Transmit DMA address.
0x014	RW	0x00000000	DMA_TX_COUNT	Transmit DMA counter.
0x018	RW	0x00000000	DMA_RX_ADDR	Receive DMA address.
0x01c	RW	0x00000000	DMA_RX_COUNT	Receive DMA counter.
0x020	R	impl def	FIFO_SIZE	The size of the FIFO.

3.5. Keyboard Controller

3.5.1. Description

This device receives keyboard button press/release events. One event is generated for each press or release action. The low 8 bits of the value is the PC scan code identifying the key. The most significant bit (bit 31) is zero for key-down events, and set for key-up events.

A maskable interrupt is raised when the event FIFO is not empty.

3.5.2. Device Properties

`compatible` `syborg,keyboard`

`fifo-size` The maximum number of events the FIFO can hold (optional, default 16).

3.5.3. Registers

Table 3.4. Keyboard Controller Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d1002	ID	Peripheral ID.
0x004	R	0xffffffff	DATA	Reads and removes the next event from the FIFO, or 0xffffffff if the FIFO is empty.
0x008	R	0x00000000	FIFO_COUNT	The number of events currently in the FIFO.
0x00c	RW	0x00000000	INT_ENABLE	`0` Interrupt masked. `1` Interrupt enabled.
0x010	R	impl def	FIFO_SIZE	The size of the FIFO.

3.6. Interval Timer

3.6.1. Description

This device is a single countdown timer. When the timer reaches zero, an interrupt is raised. This interrupt is sticky (remains set until manually cleared), and can be masked.

In periodic mode the counter is then reloaded with the LIMIT value, and continues counting.

In one-shot mode the timer stops, and the count value must be reset before the timer is re-enabled.

3.6.2. Device Properties

`compatible` `syborg,timer`

`frequency` Counter frequency in Hz.

3.6.3. Registers

Table 3.5. Interval Timer Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d1003	ID	Peripheral ID.
0x004	RW	0x00000000	RUNNING	`0` Timer stopped. `1` Timer running.
0x008	RW	0x00000000	ONESHOT	`0` Periodic. Timer count is reloaded from LIMIT when it reaches zero. `1` One-shot. Timer stops when it reaches zero.
0x00c	RW	0x00000000	LIMIT	Set the reload value. Also sets the current timer value.
0x010	RW	0x00000000	VALUE	Current timer countdown value.
0x014	RW	0x00000000	INT_ENABLE	`0` Interrupt masked.

Offset	Access	Reset	Name	Description
				`1` Interrupt enabled.
0x018	RW	0x00000000	INT_STATUS	Write 0x00000001 to this register to clear the interrupt. Read gives the interrupt state before masking (0 = clear, 1 = set).
0x01c	R	impl def	FREQ	Timer frequency in Hz.

3.7. Real-Time Clock

3.7.1. Description

This device is a free-running reference clock.

The clock counts the number of nanoseconds (10^{-9}s) since the UNIX Epoch (00:00:00 UTC, January 1, 1970).

For convenience the counter can also be read in seconds, microseconds, and milliseconds.

Because the counter value is 64-bit, accesses are explicitly latched. Issuing a read latch command (0-3) copies the current counter value to the data register. A write latch command (4) sets the counter to the value in the data register.

3.7.2. Device Properties

```
compatible  syborg,rtc
```

3.7.3. Registers

Table 3.6. Real-Time Clock Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d1004	ID	Peripheral ID.
0x004	W	N/A	LATCH	Timer latch control. `0` Read counter (nanoseconds). `1` Read counter (microseconds). `2` Read counter (milliseconds). `3` Read counter (seconds). `4` Write counter.
0x008	RW	0x00000000	DATA_LOW	The low (least significant) 32 bits of the data register.
0x00c	RW	0x00000000	DATA_HIGH	The high (most significant) 32 bits of the data register.

3.8. Pointer Controller

3.8.1. Description

This device receives events from a pointing device (mouse or touchscreen). Each event includes several button states, a Z distance (typically a scroll wheel) and XY coordinates. For mice the XY coordinates are a relative distance, while for a touchscreen they are absolute coordinates. Pending events are stored in a FIFO.

A maskable interrupt is raised when the event FIFO is not empty.

3.8.2. Device Properties

`compatible` `syborg,pointer`

`absolute` 0 = mouse, 1 = touchscreen. (optional, default 1).

`fifo-size` The maximum number of events the FIFO can hold (optional, default 16).

3.8.3. Registers

Table 3.7. Pointer Controller Registers

Offset	Access	Reset	Name	Description
0x000	R	impl def	ID	Peripheral ID. 0xc51d0005 for mouse, 0xc51d0006 for touchscreen.
0x004	W	N/A	LATCH	Write 1 to this register to load the next FIFO entry. Has no effect if the FIFO is empty.
0x008	R	0x00000000	FIFO_COUNT	The number of events currently in the FIFO.
0x00c	R	0x00000000	X	Current X coordinate. For mice this is a signed delta from the previous position. For touchscreens it is an absolute value between 0 and 0x7fff. The positive X axis is to the right.
0x010	R	0x00000000	Y	Current Y coordinate. For mice this is a signed delta from the previous position. For touchscreens it is an absolute value between 0 and 0x7fff. The positive Y axis is downwards.
0x014	R	0x00000000	Z	Current Z coordinate. This is a signed delta from the previous value. The positive Z axis is downwards.
0x018	R	0x00000000	BUTTONS	Each bit indicates the state of a button. Buttons are numbered from the least significant bit. A nonzero bit indicates that the button is pressed.
0x01c	R	0x00000000	INT_ENABLE	`0` Interrupt masked. `1` Interrupt enabled.
0x020	R	impl def	FIFO_SIZE	The size of the FIFO.

3.9. Framebuffer

3.9.1. Description

This device is a framebuffer that has the following capabilities:

- Different bpp depths (1, 2, 4, 8, 15, 16, 24, 32)
- Color palette for bpp depths 1, 2, 4, 8
- LE/BE byte ordering
- LE/BE pixel ordering
- RGB/BGR color ordering (for 16 and 32 bpp modes)
- Resizeable screen
- Fast blank screen mode
- Interrupt raising
- Variable orientation: 90-degree stepped rotation, plus flipping
- Non-consecutive row arrangement

3.9.2. Configuring the Framebuffer device

Configuration can be performed with the device in disabled state (FB_ENABLED = 0) to avoid unexpected behavior from a half-configured device.

Sample configuration sequence:

1. Disable the device (FB_ENABLED = 0)
2. Specify the base address memory where the data will be read (FB_BASE = addr)
3. Specify the interrupts mask (FB_INT_MASK)
4. Specify the rest of the configuration settings (i.e. height, width, orientation, palette)
5. Enable the device (FB_ENABLED = 1)

FB_BASE update (for page flipping) and palette changes can safely be performed without disabling the device.

3.9.3. Pixel Formats

The framebuffer interprets framebuffer data by loading 32-bit words of data from RAM, splitting each word into pixels, then interpreting each pixel as either an index into the palette or a set of 3 color components. The exception is 24-bit mode where each pixel is fetched as 3 bytes. See individual registers for full details.

3.9.4. Device Properties

```
compatible  syborg,framebuffer
```

`width` Default width in pixels (optional)

`height` Default height in pixels (optional)

3.9.5. Registers

Table 3.8. Framebuffer Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d1007	ID	Peripheral ID.
0x004	RW	0x00000000	BASE	Base memory address where the data will be read. The address must be aligned to a 4-byte boundary.
0x008	RW	impl def	HEIGHT	Screen height (in pixels).
0x00c	RW	impl def	WIDTH	Screen width (in pixels). The length of the resulting row data must be a multiple of 4 bytes. e.g. for 16 bpp mode this must be a multiple of 2.
0x010	RW	0x00000000	ORIENTATION	Set display orientation. `0` No rotation. `1` 90 degree counterclockwise rotation. `2` 180 degree rotation. `3` 90 degree clockwise rotation. `4` Vertical flip. `5` 90 degree counterclockwise rotation followed by vertical flip. `6` Horizontal flip. `7` 90 degree clockwise rotation followed by vertical flip. **Note** This has no direct affect on the operation of the framebuffer. It is simply a hint how the resulting image should be displayed.
0x014	RW	0x00000000	BLANK	Blank screen mode. Causes the device to display a black screen, ignoring the contents of the framebuffer data. The device does not trigger the VSYNC interrupt during the Blank screen mode. `0` Normal operation. `1` Blank screen.

Offset	Access	Reset	Name	Description
0x018	RW	0x00000000	INT_MASK	Integer bit mask controlling which interrupts are active. It is an OR'ed combination of the following bits: `0x01` Enables the VSYNC interrupt. This interrupt triggers immediately after a screen refresh. `0x02` Enables the BASE_UPDATE_DONE interrupt. Raised at the same time as VSYNC, but only if the FB_BASE register has been modified since the last refresh. **Note** Screen refreshes may occur at unpredictable intervals, so the VSYNC interrupt should not be used to control timing.
0x01c	RW	0x00000000	INT_CAUSE	This register is an integer bit mask, where each bit represents the asserted interrupt. Read and write this register in the interrupt handling routine. Each bit set will clear the corresponding interrupt. Refer to the FB_INT_MASK register for the possible bit values and the description of each interrupt.
0x020	RW	0x00000020	BPP	Bits per pixel. Valid values are as follows: `1` 2-color palette. `2` 4-color palette. `4` 16-color palette. `8` 256-color palette. `15` High bit ignored. 5 bits each for red, green and blue. `16` 5 bits red, 6 bits green, 5 bits blue. `24` 8 bits each for red, green and blue. `32` 8 high bits ignored. 8 bits each for red, green and blue.
0x024	RW	0x00000000	COLOR_ORDER	Order of the red, green, and blue components. `0` BGR (blue is LSB, red is MSB). `1` RGB (red is LSB, blue is MSB)
0x028	RW	0x00000000	BYTE_ORDER	Endianness of the framebuffer source data. Data is fetched in 32-bit words, and the whole word is byte

Offset	Access	Reset	Name	Description
				swapped according to this register, independent of the pixel size. This is ignored for 24 bpp mode. `0` Little-endian `1` Big-endian
0x02c	RW	0x00000000	PIXEL_ORDER	Controls how multiple pixels are packed into a byte. This only affects 1, 2 and 4 bpp modes. The least significant byte of the word always contains the first 8/4/2 pixels. `0` Little-endian (first pixel is LSB). `1` Big-endian (first pixel is MSB).
0x030	RW	0x00000000	ROW_PITCH	Number of bytes between the start of consecutive rows. If set to zero then this register is ignored and the pitch is the length of the row. Must be a multiple of 4.
0x034	RW	0x00000000	ENABLED	Enables or disables the framebuffer device, allowing configuration changes during the disabled mode to avoid unexpected behavior from a half configured device. `0` Device is disabled. `1` Device is enabled.
0x400-0x7fc	RW	0x00000000	PALETTE	Palette values to use in 1, 2, 4, and 8 bpp modes. There is one 32-bit word per entry. The color encoding is: Bits 31-24 Unused. Bits 23-16 Red. Bits 15-8 Green. Bits 7-0 Blue.

3.10. Host Filesystem Device

3.10.1. Description

This device provides access to the host filesystem.

Access is implemented via a virtual syscall interface. Syscall arguments are loaded into the corresponding device argument registers. The Syscall is triggered by writing the syscall number to the `COMMAND` register. The syscall completes and the results are ready immediately. The `RESULT` register

is zero if the syscall succeeded and a negative error value if the syscall failed. Other values may be returned in the device argument registers.

Filenames are specified using a pair of argument values. The first value is a memory address locating the start of the string, and the second is a length. Filenames are encoded using 16-bit Unicode characters (a.k.a. UTF-16 or UCS-2). The filename length is the number of 16-bit characters. The filename need not be null terminated.

All filenames should include absolute paths including the drive specifier. (e.g. `N:\TESTS\LOG.TXT`). The device has no concept of a current working directory, and relative paths are not supported. A backslash (`\`) is used as a directory separator.

The supplied filename is mapped onto the host system by removing the drive specifier and prepending a host directory.

Warning

This device does not provide a secure sandbox environment. It may be possible to access files outside the specified host directory.

Warning

Care should be taken to avoid modifying a file or directory on the host while it is in use by the virtual machine. No facilities are provided for synchronisation or locking between different machines.

3.10.2. Syscalls

The following syscalls are available:

MkDir (1) — Create a new directory. `ARG0/ARG1` is the name of the new directory.

RmDir (2) — Remove a directory. `ARG0/ARG1` is the name of the directory to be removed. The directory must be empty.

Delete (3) — Remove a file. `ARG0/ARG1` is the name of the file to be removed.

Rename (4) — Rename a file. `ARG0/ARG1` is the name of an existing file, and `ARG2/ARG3` is the new name for the file.

Replace (5) — Move a file, replacing another if necessary. `ARG0/ARG1` is the name of an existing file, and `ARG2/ARG3` is the new name for the file.

Get Entry (7) — Query attributes of a file. `ARG0/ARG1` is the name of the file to query. Upon successful completion the following values are set:

`ARG0`	File attributes. A combination of the following bit flags: `0x01` Readonly `0x02` Hidden `0x10` Directory

ARG1	File last modification time. This is the number of seconds from the UNIX Epoch (00:00:00 UTC, January 1, 1970).
ARG2	File size in bytes.

Open File (9)

Open a file. ARG0/ARG1 is the name of the file to open. Upon successful completion the following values are set:

ARG0	The newly opened file handle.
ARG1	File attributes. A combination of the following bit flags: 0x01 Readonly 0x02 Hidden 0x10 Directory
ARG2	File last modification time. This is the number of seconds from the UNIX Epoch (00:00:00 UTC, January 1, 1970).
ARG3	File size in bytes.

Open Directory (10)

Open a directory listing. ARG0/ARG1 is the filename pattern to match. The last filenename component may include * and ? wildcards. If the pattern ends in \ then it is equivalent to *. ARG0 is set to the newly opened directory handle.

Close File (11)

Close a file handle. ARG0 is the file handle to close.

Read From File (12)

Read data from a file. ARG0 is the file handle to read from. ARG1 is the offset from the start of the file. ARG2 is the memory address to write the read data to. ARG3 is the number of bytes to read. Upon successful completion ARG0 is set to the number of bytes read. This may be less than the requested size if the end of the file is reached.

Write To File (13)

Write data to a file. ARG0 is the file handle to write to. ARG1 is the offset form the start of the file. ARG2 is the memory address to read the read data from. ARG3 is the number of bytes to write. Upon successful completion ARG0 is set to the number of bytes written.

Set File Size (14)

Truncate a file. ARG0 is the file handle to be truncated. ARG1 is the new length of the file, in bytes.

Flush (15)

Flush any buffered file data to underlying storage. ARG0 is the file handle to flush.

Close Directory (16)

Close a directory handle. ARG0 is the directory handle.

Read Directory (17)

Read a directory entry. To obtain a complete list of files this this should be called repeatedly until an EOF error is encountered. ARG0 is the directory handle to read from. ARG1 is the memory address of the buffer to receive the name. ARG2 is the size (in 16-bit characters) of the buffer pointed to by ARG1. Upon successful completion the name of the directory entry is written into the buffer, and the following values are set:

`ARG0`	File attributes. A combination of the following bit flags: `0x01` Readonly `0x02` Hidden `0x10` Directory
`ARG1`	File last modification time. This is the number of seconds from the UNIX Epoch (00:00:00 UTC, January 1, 1970).
`ARG2`	File size in bytes.
`ARG3`	The length of the name (in 16-bit characters).

3.10.3. Errors

The following result codes may be returned:

Error	**Value**
None (Success)	0
NotFound	-1
General	-2
NoMemory	-4
NotSupported	-5
InvalidArgument	-6
BadHandle	-8
AlreadyExists	-11
PathNotFound	-12
InUse	-14
Unknown	-19
Corrupt	-20
AccessDenied	-21
Locked	-22
Write	-23
Eof	-25
DiskFull	-26
BadName	-28
Abort	-39
TooBig	-40
DirFull	-43
PermissionDenied	-46

3.10.4. Device Properties

`compatible` `syborg,hostfs`

`host-path` The host path that specifies the root directory for this device.

`drive-number` The number to identify this device. Typically drive number 1 is `A:`, drive number 2 is `B:`, etc.

3.10.5. Registers

Table 3.9. Host Filesystem Device Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d0008	ID	Peripheral ID.
0x004	RW	0x00000000	COMMAND	Writing a value to this address causes the specified command to be executed.
0x008	RW	0x00000000	RESULT	The result of the last command. A value of zero indicates success, a negative error code indicates failure.
0x00c	RW	0x00000000	ARG0	The first argument register.
0x010	RW	0x00000000	ARG1	The second argument register.
0x014	RW	0x00000000	ARG2	The third argument register.
0x018	RW	0x00000000	ARG3	The fourth argument register.

3.11. Snapshot Device

3.11.1. Description

This device allows snapshots of the virtual machine to be created and restored. The name of the snapshot is an ASCII string read from RAM. The name need not be null terminated.

> **Note**
>
> The snapshot might not take effect immediately. It can occur any time between the `TRIGGER` store and the next branch instruction.

Normally QEMU stores machine snapshots in a QCOW2 image file, along with a snapshot of the image. If the snapshot name starts with `file:` then the rest of the name is interpreted as a filename, and the snapshot is read from/written to that file.

Snapshots can also be loaded with the `-loadvm` *`name`* commandline option or the `loadvm` monitor command.

> **Warning**
>
> The host filesystem device allows a machine to interact with files not under the control of QEMU. The state of these files is not included in the snapshot, and any open handles will be closed when the snapshot is restored.

3.11.2. Device Properties

```
compatible  syborg,snapshot
```

3.11.3. Registers

Table 3.10. Snapshot Device Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d0009	ID	Peripheral ID.
0x004	RW	0x00000000	ADDRESS	The memory address of the snapshot name.
0x008	RW	0x00000000	ADDRESS	The length (in bytes) of the snapshot name.
0x00c	W	N/A	TRIGGER	Write 1 to this register to create a snapshot. Write 2 to this register to restore from a snapshot.

3.12. Network Device

3.12.1. Description

This device provides a virtual Ethernet network device. It is based on the Linux virtio-net interface. For details of the Virtio interface see Chapter 4, “Virtio”.

Two Virtio queues are used. The first to receive incoming packets, and the second to transmit outgoing packets.

This first 256 bytes of device address space is control registers, as with other devices. Offsets from 0x100 onwards provide access to the virtio-net config space, and accepts accesses of any size.

3.12.2. Sending Packets

Each request should be start with a 10 byte header. This header is not currently used, and should be zeroed. The remainder of the request constitutes the packet data.

3.12.3. Receiving Packets

Requests in the receive queue should allocate space for a 10 byte header. When a packet arrives the device will fill in the header, and copy the packet data to the remainder of the request. The header is currently unused and should be ignored.

3.12.4. Device Properties

```
compatible  syborg,virtio-net
```

3.12.5. Registers

Table 3.11. Network Device Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d000a	ID	Peripheral ID.
0x004	R	0x00000001	DEVTYPE	Virtio device ID.
0x008	R	impl def	HOST_FEATURES	For future expansion. Allows the device to expose a set of feature bits.

Offset	Access	Reset	Name	Description
0x00c	RW	0x00000000	GUEST_FEATURES	For future expansion. Allows the driver to acknowledge support for a set of features.
0x010	RW	0x00000000	QUEUE_BASE	Set the address of the selected virtqueue. Should be aligned on a 4k boundary.
0x014	R	impl def	QUEUE_NUM	The size of the selected virtqueue ring. Will be zero for unimplemented queues.
0x018	RW	0x00000000	QUEUE_SEL	Select which virtqueue is accesses by the `QUEUE_BASE` and `QUEUE_NUM` registers. Writing a value of 0 selects is the first queue and writing 1 selects the second.
0x01c	W	N/A	QUEUE_NOTIFY	Notify the device that new requests have been placed in a virtqueue. The value written determines which queue is checked.
0x020	RW	0x00000000	STATUS	Set Virtio device status bits. This is used to inform the device about the state of the OS driver. The device will not begin servicing requests until the device driver indicates that it is ready. A combination of the following bitflags should be used: 1 The device has been detected. 2 A driver has been associated with the device. 4 The device driver has completed initialization and it read for operation. Writing zero to this register resets the device.
0x024	RW	0x00000000	INT_ENABLE	`0` Interrupt masked. `1` Interrupt enabled.
0x028	RW	0x00000000	INT_STATUS	Write 0x00000001 to this register to clear the interrupt. Reads will have the low bit set if the notification interrupt is pending.
0x100	RW	impl def	config	The device config is accessible from this offset. The first 6 bytes of the device config can be read to determine the device MAC address.

3.13. NAND Flash Device

3.13.1. Description

This provides NAND flash based storage. The virtual device emulates a connection to a Samsung NAND flash chips such as the K9F2808U0A.

A backing file for the contents of the flash device can be specified using the `-drive if=mtd,file=`*`filename.img`* commandline option. The size of the file is used to determine whether it includes the OOB data. e.g. for a 128Mbit device a 16Mbyte file is treated as 512 byte blocks, with the additional 16 "spare" bytes per block being held in memory, and initially zeroed. A

16.5Mbyte file is treated as a set of 528 byte blocks. If no backing file then all the data is help in memory and discarded when the virtual machine terminates.

The device also contains an ECC engine to assist with software error detection and correction. This maintains rolling parity bits based on the values read from and written to the DATA register.

Parity bits are calculated as follows:

P1' bit0 ^ bit2 ^ bit4 ^ bit6 ^ bit8 ^ bit10 ^ bit12 ^ bit14 ...

P1 bit1 ^ bit3 ^ bit5 ^ bit7 ^ bit9 ^ bit11 ^ bit13 ^ bit15 ...

P2' bit0 ^ bit1 ^ bit4 ^ bit5 ^ bit8 ^ bit9 ^ bit12 ^ bit13 ...

P2 bit2 ^ bit3 ^ bit6 ^ bit7 ^ bit10 ^ bit11 ^ bit14 ^ bit15 ...

P4' bit0 ^ bit1 ^ bit2 ^ bit3 ^ bit8 ^ bit9 ^ bit10 ^ bit11 ...

P4 bit4 ^ bit5 ^ bit6 ^ bit7 ^ bit12 ^ bit13 ^ bit14 ^ bit15 ...

etc.

The following devices are supported:

Size (Mbit)	Chip ID	Page Size (bytes)	Erase Size (pages)
1	0x6e	256	16
2	0x64	256	16
4	0x6b	512	16
8	0xd6	512	16
16	0x33	512	32
32	0x35	512	32
64	0x36	512	32
128	0x78	512	32
256	0x71	512	32
512	0xa2	2048	64
1024	0xa1	2048	64
2048	0xaa	2048	64
4096	0xac	2048	64
8192	0xa3	2048	64
16384	0xa5	2048	64

All devices are 8 bits wide.

3.13.2. Device Properties

`compatible` `syborg,nand`

`size` The size of the flash device in Mbits.

3.13.3. Registers

Table 3.12. NAND Flash Device Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d000b	ID	Peripheral ID.
0x004	RW	impl def	DATA	The low 8 bits of this register map onto the 8 Data Input/Output pins on the flash chip. The high 24 bits are ignored.
0x008	RW	0x00000000	CTL	Read or write flash chip control pins. Bit mapping as as follows: Bit0 Command Latch Enable. Bit1 Address Latch Enable. Bit2 Chip enable (active low). Bit3 Write Protect (active low). Bit4 Read(1)/Busy(0) Output. Writes to this bit are ignored.
0x00c	RW	0x00000000	ECC_COUNT	The number of bytes processed by the ECC engine. Write zero to this register to reset the ECC engine.
0x010	RW	0x00000000	ECC_CP	ECC column parity. Bit mappings as follows: Bit0 P1' Bit1 P1 Bit2 P2' Bit3 P2 Bit4 P4' Bit5 P4 Bits 6-31 are unused and read as zero.
0x014	RW	0x00000000	ECC_LP	ECC line parity. Bit0=P8', Bit1=P8, Bit2=P16', Bit3=P16, etc.

3.14. Audio Device

3.14.1. Description

This device provides audio playback and recording. It is based on the Linux virtio-net interface. For details of the Virtio interface see Chapter 4, “Virtio”.

Three Virtio queues are used. The first queue is a command queue, used to configure the device. The remaining queues form two independent audio streams. Each stream can be configured for either output (playback) or input (recording).

Each stream queue transfers sample data to/from the associated stream. Sample data is organised in frames, with each frame containing one sample for each channel.

This first 256 bytes of device address space is control registers, as with other devices. Offsets from 0x100 onwards provide access to the virtio-net config space, and accepts accesses of any size.

3.14.2. Configuration Commands

The device is configured by placing commands in the command queue. A command is 12 bytes long, represented by the following structure:

```
struct virtio_audio_cmd
{
  uint32_t command;
  uint32_t stream;
  uint32_t arg;
};
```

Each command applied to a single stream. The meaning of the argument is dependent on the command. Multiple commands may be submitted in a single request. Some commands write a result value, others simply effect stream operation.

Table 3.13. Audio Device commands

Command	value	Description
Set Endian	1	Set the endianess of sample data. arg=0 Little endian arg=1 Big endian
Set Channels	2	Set the number of channels. 1 Mono 2 Stereo
Set Format	3	Set the sample format. 0 Unsigned 8-integer 1 Signed 8-bit integer 2 Unsigned 16-integer 3 Signed 16-bit integer 4 Unsigned 32-integer 5 Signed 32-bit integer
Set Frequency	4	Set the number of frames per second.

Command	value	Description
Init	5	Prepare a stream for operation. This should be done after setting the sample format and frequency, and before setting the stream running. 0 Stream is receiving data (capture) 1 Stream is sending data (playback) This command returns a 32-bit integer specifying the size in bytes of the hardware buffer associated with this stream. This value may be zero if the size is not known.
Run	6	Start for stop stream operation. 0 Stop stream 1 Start stream

3.14.3. Device Properties

`compatible  syborg,virtio-audio`

3.14.4. Registers

Table 3.14. Audio Device Registers

Offset	Access	Reset	Name	Description
0x000	R	0xc51d000a	ID	Peripheral ID.
0x004	R	0x0000ffff	DEVTYPE	Virtio device ID.
0x008	R	impl def	HOST_FEATURES	For future expansion. Allows the device to expose a set of feature bits.
0x00c	RW	0x00000000	GUEST_FEATURES	For future expansion. Allows the driver to acknowledge support for a set of features.
0x010	RW	0x00000000	QUEUE_BASE	Set the address of the selected virtqueue. Should be aligned on a 4k boundary.
0x014	R	impl def	QUEUE_NUM	The size of the selected virtqueue ring. Will be zero for unimplemented queues.
0x018	RW	0x00000000	QUEUE_SEL	Select which virtqueue is accesses by the `QUEUE_BASE` and `QUEUE_NUM` registers. Writing a value of 0 selects is the first queue and writing 1 selects the second.
0x01c	W	N/A	QUEUE_NOTIFY	Notify the device that new requests have been placed in a virtqueue. The value written determines which queue is checked.
0x020	RW	0x00000000	STATUS	Set Virtio device status bits. This is used to inform the device about the state of the OS driver. The device will not begin servicing requests until the device driver indicates that it is ready. A combination of the following bitflags should be used:

Offset	Access	Reset	Name	Description
				1 The device has been detected. 2 A driver has been associated with the device. 4 The device driver has completed initialization and it read for operation. Writing zero to this register resets the device.
0x024	RW	0x00000000	INT_ENABLE	0 Interrupt masked. 1 Interrupt enabled.
0x028	RW	0x00000000	INT_STATUS	Write 0x00000001 to this register to clear the interrupt. Reads will have the low bit set if the notification interrupt is pending.
0x100	RW	impl def	config	The device config is accessible from this offset. The first bytes the device config can be read to determine the number of streams implemented by the device.

Chapter 4
Virtio

Several base platform devices are built on top of the Virtio framework for virtual IO devices. This chapter describes that framework.

4.1. Introducing Virtio

The Virtio framework was developed to allow the Linux kernel to access virtual IO devices provided by a variety of hypervisors. It provides a common interface for interacting with a variety of different IO devices.

The majority of communication with a device is done via virtqueues. A virtqueue is a queue of buffers provided by the guest for consumption by he host. Each buffer may contain several readable or writable parts, located by a scatter-gather array.

Authorative documentation for Virtio devices can be found in the Linux kernel sources and Russell, R. 2008. virtio: towards a de-facto standard for virtual I/O devices. SIGOPS Oper. Syst. Rev. 42, 5 (Jul. 2008), 95-103. DOI= http://doi.acm.org/10.1145/1400097.1400108

4.2. Virtio_ring

SVP devices implement virtqueues using a simple ring buffer interface known as virtio_ring.

Each virtio_ring consists of 3 parts: An array of descriptors used to implement scatter-gather lists, immediately followed by an avail ring used to submit requests to the device. The used ring allows the device to report request completion, and is located starting on the next 4k page boundary after the avail ring.

The descriptor array and avail ring are managed and written by the guest OS driver. The used ring is written by the device and read by the OS driver.

The format of a descriptor is as follows:

```
struct vring_desc
{
  uint64_t addr;
  uint32_t len;
  uint16_t flags;
  uint16_t next;
};
```

The `addr` and `len` fields identify and area of memory. The next field is used to chain multiple descriptors into a single request. The flags field is a combination of the following flags:

`0x0001` (Next) — If set then the `next` field contains the index of the next descriptor for this request. If clear then this is the last descriptor in the request.

`0x0002` (Write) — If set then this section is to be written by the device. If clear then this section is to be read by the device.

The format of the avail ring is as follows:

```
struct vring_avail
{
  uint16_t flags;
  uint16_t idx;
  uint16_t ring[NUM];
};
```

The `idx` field is a free running index identifying the head of the submission queue. It indexes (with wrapping) into a ring of descriptor indexes. This separation of request submission and descriptor management allows the virtqueue to operate in an asynchronous manner, and prevents long running requests from blocking short requests.

A notification will be generated when the device reads a request from the avail ring. Setting the low bit of the `flags` field will suppress these notifications.

The format of the used ring is as follows:

```
struct vring_used
{
  uint16_t flags;
  uint16_t idx;
  struct {
    uint16_t id;
    uint16_t len;
  } ring[NUM];
};
```

The used ring works much the same way as the the avail ring. When a request completes the index of the first descriptor and the number of bytes is written into the next used ring entry, and the `idx` field is incremented. The number of bytes processed may be less than the total submitted. e.g. a network device may write one packet per request, even if a single request is large enough to hold multiple packets.

A notification is generated when the device reads a request from the avail ring. Setting the low bit of the `flags` field suppresses this notifications.

To submit a new request the OS driver should first construct a descriptor chain for the request. Then it should write the index of the first descriptor into the next slot in the avail ring. Once this has been completed the `idx` field in the avail ring should be incremented. Finally the device should be notified that new requests are available. Multiple requests may be added before performing the notification.

When the request completes the device will write write it to the next entry in the used ring, then increment the `idx` field.

Descriptors must not be reused until this has occurred. Some devices may process and complete requests out of order. Provided there are unused descriptors available, Additional requests may be submitted to the avail ring even if previous requests have not completed. Requests can not be cancelled or removed once they have been submitted to the device.

A notification is generated when a request completes and is added to the used ring. Setting the low bit of the `flags` field suppresses this notifications.

4.3. Virtual Platform Bindings

The base Symbian Virtual Platform does not implement a PCI bus, so the normal Linux device binding cannot be used. Instead a very similar interface is implemented via a set of memory mapped registers.

These registers are described in the individual device descriptions.

Chapter 5
Device Plugins

The Symbian Virtual Platform provides a device plugin mechanism. This allows third party devices (including emulation of real hardware devices) to be added to the virtual machine.

5.1. Introduction

The Symbian Virtual Platform includes a plugin mechanism that allows emulation of additional devices.

Device plugins are loaded as Python modules. These can be implemented directly in Python, or as a Python wrapper around some other language (typically C/C++).

Device modules can access to QEMU functionality by importing the `qemu` module.

5.2. Device Classes

A device class corresponds to a particular type of device. Device classes are implemented as a Python class derived from qemu.devclass. This class type is then used to instantiate individual devices.

A device class should be derived from `qemu.devclass`. New device classes can be registered by calling `qemu.register_device(newclass)`.

The class should define the following attributes:

`irqs` The number of IRQ outputs used by the device.

`regions` A list of `qemu.ioregion` objects describing memory mapped IO regions provided by the device. Each entry in the list corresponds to an entry in the `reg` machine description property.

Each `qemu.ioregion` object has the following attributes:

- `size` The size of the region, in bytes. This should be a power of two.
- `readl` This function will be called when a read operation occurs. This should be a method of the containing class. The method will be called with the offset from the start of the region, and should return the value read.
- `writel` This function will be called when a write operation occurs. This should be a method of the containing class. The method will be called with the offset from the start of the region and the value to be written.

`name` The name of the device. This is matched against the `compatible` machine description property.

`properties` A dictionary of {`name`:`value`} pairs. Each name is matched against properties in the machine description. The value specifies the default value for the property, and the property type (string or integer) is inferred from the default value. When a device is instantiated the `properties` attribute of the device object will be populated with values from the machine description. If the machine description does not specify a property then the default will be used.

The `chardev` property is special. If present this should be given a default value of `None`. This will be replaced with a `qemu.chardev` object when the device is instantiated.

Device classes should override the following methods:

`create(self)`	Initialize the device. The `properties` attribute is populated with values from the machine description before this method is called. Any attributes or data required by the device should be added in this function.
`save(self, state)`	Save device state.
`load(self, state)`	Restore device state from state.

Warning

The `__init__` method should not be overridden.

5.3. Device Object Methods

Device objects provide the following methods to assist with implementing device emulation:

`set_irq_level(irq, level)`	Rise or lower a device IRQ output. `irq` is a zero based index specifying which IRQ to modify. `level` is zero to lower the IRQ or one to raise it.
`create_interrupts(callback, count)`	Create input IRQ lines (for interrupt controller type devices). When the IRQ line state changes the function `callback(irq, level)` will be called.
`dma_readb(address)`	Read a byte from system memory.
`dma_writeb(address, value)`	Write a byte to system memory.

5.4. Snapshots

In order to support snapshots, devices must be able to save and restore internal state. The is achieved by overriding the `save` and `load` methods. The snapshot state can be accessed via the object passed to these methods. This state object provides the following methods:

`get_u32()`	Read an unsigned 32-bit integer.
`get_u64()`	Read an unsigned 64-bit integer.
`get_s64()`	Read a signed 64-bit integer.
`put_u32()`	Write an unsigned 32-bit integer.
`put_u64()`	Write an unsigned 64-bit integer.
`put_s64()`	Write a signed 64-bit integer.

5.5. Timers

The `qemu.ptimer` class provides countdown timer functionality. When the timer reaches zero a function is called. In one-shot mode the timer is then disabled. In periodic mode the counter is reloaded and immediately resumes counting down.

Functions are also provided to query the current virtual clock.

`qemu.get_clock()`	Return the current value of the high resolution virtual clock. This is the number of nanoseconds since the machine was created.
`qemu.start_time()`	Return the time when the machine was created. This is the number of seconds since the Unix Epoch (00:00:00 UTC, January 1, 1970).
`qemu.ptimer(tick, freq)`	Create a new timer counting at the specified frequency (Hz). The function `tick()` will be called when the timer expires.
`ptimer.count`	The current counter value.
`ptimer.run(oneshot)`	Start the timer in either one-shot (True) or periodic (False) mode. **Note** This does not reset the counter value. In one-shot mode the counter must be explicitly reloaded after it reaches zero.
`ptimer.stop()`	Stop the timer.
`ptimer.set_limit(limit, reload)`	Set the reload value when in periodic mode. This has no effect in one-shot mode. If `reload` is True then also set the counter value to `limit`.
`ptimer.get(state)`	Restore state from snapshot.
`ptimer.put(state)`	Save state to snapshot.

5.6. Keyboard

The `qemu.register_keyboard(handler)` function allows a device to respond to keyboard input. When a keyboard event occurs `handler(keycode)` is called. The keycode is a standard PC scancode. Extended scanodes are prefixed by an 0xe0 byte, and bit 7 (0x80) of the scancode indicates whether this is a press (clear) or release (set) event.

5.7. Mouse

The `qemu.register_mouse(handler, absolute)` function allows a device to respond to mouse input. When a mouse event occurs `handler(x, y, z, buttons)` is called. The `absolute` argument should be True for touchscreen type devices, and False for mouse type devices.

The `x` and `y` values specify the horizontal and vertical cursor position. In absolute mode these are coordinates between 0 (top/left) and 32767 (bottom/right). In relative mode they are distance from the previous position.

The `z` value specifies the movement of the scroll wheel.

Bits 0,1 and 2 of the `buttons` value specify the state of the left, right and middle mouse buttons respectively. A set bit indicates that the button is pressed.

5.8. Interfacing with C code

Device plugins may be implemented in any language (e.g. C/C++). However in order to do this a Python interface to this code must be created. The simplest way to do this is to use SWIG (http://www.swig.org/) to generate the interface.

SWIG uses an interface file to generate the Python interface code. In simple cases this interface file is very similar to the C header file used to declare the functions.

This is best demonstrated by a simple example. Consider the following C code (`myfoo.c`):

```
int myfoo(int a, int b)
{
 return a * b;
}
```

The SWIG interface file (`foo.i`) for this is as follows:

```
%module foo
%{
int myfoo(int a, int b);
%}
int myfoo(int a, int b);
```

The interface file consists of two parts. The section inside `%{`/`%}` is used to generate the interface code, and the second declaration is used to provide C prototypes and inline code for use by this interface code. `#include` directives can be used to avoid this duplication.

The following command generates the Python interface code for this module:

```
swig -python foo.i
```

This command creates two files, `foo.py` and `foo_wrap.c`. `foo.py` provides the Python part of the module. `foo_wrap.c` should be linked with your C code to provide the actual implementation module as follows:

```
gcc -shared -o _foo.pyd foo_wrap.c myfoo.c \
-I qemu-installdir\include qemu-installdir\lib\python26.dll
```

The files `foo.py` and `_foo.pyd` should then be copied to `share\qemu\plugins` so that they can be found by other plugins.

To use this module from a Python plugin it should first be imported with `import foo`. It can then be used the same as any other Python module, for example `print foo.myfoo(2, 3)` will print `6`.

For more details about using SWIG, including more complicated examples and how to call Python routines from C, see see http://www.swig.org/Doc1.1/HTML/Python.html.