The Report Committee for Evan Robert Platt
Certifies that this is the approved version of the following report:

Virtual Peripheral Interfaces in

Emulated Embedded Computer Systems

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor:

Vijay K. Garg

Al Williams

Virtual Peripheral Interfaces in
Emulated Embedded Computer Systems

by

EVAN ROBERT PLATT, B.S.E.E.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

DECEMBER 2016

Abstract

Virtual Peripheral Interfaces in
Emulated Embedded Computer Systems

Evan Robert Platt, M.S.E.

The University of Texas at Austin, 2016

Supervisor: Vijay K. Garg

Small form-factor single-board computers (SBCs) have become a popular platform chosen by hobby and professional developers to host software projects. In recent years, the Raspberry Pi has become the most popular platform available, in part due to its ability to run a full-blown Linux operating system – the same distributions available for desktop PCs. This results in greater ease of use, and a familiar software environment for users. No matter what operating system is running on the developer's PC, software to be run on the SBC can be debugged by running it under QEMU, a multi-platform emulation software. However, if the peripheral input/output pins of the SBC are to be used by the software under development, existing emulator capabilities are insufficient for debugging as they do not offer general purpose input/output (GPIO) capabilities.

This project implements a solution to GPIO debugging while using an emulated SBC – a virtual GPIO interface that is shared with the emulation's host PC. In order to make use of the virtual interface a software solution is also presented for each side of the interface – the SBC program and the peripheral emulated by the host PC. To facilitate emulation of an SBC program, a library commonly used for input/output interactions is modified to work within QEMU. To provide an example of peripheral emulation, custom LED and button widgets for the Qt user interface framework are implemented. Finally, a performance test is run to demonstrate the virtual interface's usefulness to developers.

Table of Contents

Introduction

Any software developer will affirm that it is rare for a non-trivial program to work as intended on the first execution. Methods of debugging the inevitable failures have evolved over time. In early computing, programmers had to "desk check" their software by manually reviewing source code in the hope of spotting faults. Other brute-force methods of debugging were also common, such as adding strategically-placed output messages in the source code to attempt to locate errors upon execution. Modern development environments provide a more productive approach: Interactive tools allow programmers to run code under observation of a debugger. This allows insight into the internal state of the machine and the program's execution.

When developing software for embedded targets, however, expensive tools are often required to achieve a similar level of testing capability. Even if the desired tools are available, they are often intrusive to the system under test and may alter the test results. Desktop computer-based simulation tools are sometimes available, however they often don't simulate the entire system under test. While they may be able to run the software on the emulated CPU, they often do not provide a method of emulating the system's peripheral hardware. In light of these limitations, embedded software developers are often forced to rely on the aforementioned outdated methods of debugging such as desk checking and output message placement.

The selection of PC-based simulation tools has expanded as the complexity of embedded software has increased. However, since most PC systems are built around an x86 (32 or 64 bit) system architecture, and embedded single-board computers (SBCs) are commonly based on another processor architecture, the SBC-targeted software cannot simply be executed on the developer's system. Instead, an emulated hardware system must be used to provide the correct architecture environment for the software under test.

When software is to be tested within a system emulation there is no good solution for testing GPIO functionality. Rather, the internal functions of the software can be tested within the developer's PC using the emulation, and then the software must be deployed to the SBC to

continue testing any external interfaces. The goal of this project is to implement a solution to provide a method of testing GPIO interfaces while using an emulated SBC system on a PC platform.

This project will focus on developing a solution for one SBC – the Raspberry Pi – and one emulation system – QEMU. Over the past few years, the Raspberry Pi has become a popular SBC system due to its small form factor, many on-board features, common operating system support, and low cost. Since the introduction of its first generation in 2012, over 10 million Raspberry Pi SBCs have been sold [1]. Due to this popularity, many users form on-line communities that allow developers to support one another, come up with new ideas, and share example Raspberry Pi projects. The prevalent emulation package for testing of Raspberry Pi software is QEMU, which allows emulation of the ARM architecture (used by the Raspberry Pi's CPU) on common desktop computers operating systems such as Windows, Linux, or macOS. Although it is common for developers to request help in finding a method of testing the Raspberry Pi's GPIO interfaces while running software under QEMU, no model of the GPIO functionality currently exists. Developers either do without or create ad hoc solutions to model their specific peripheral devices.

The aim of this implementation is to provide an interface between the GPIO ports of the emulated Raspberry Pi and the developer's host system environment. In addition, an example of a host-side hardware peripheral simulation will be developed to interact with the Raspberry Pi over the new interface. Several heuristics will be considered when choosing the methods employed to enable this virtual GPIO interface including the latency of the interface and the ease of adapting SBC-targeted software to use the interface during testing.

State of the Art

A quick search of any active Raspberry Pi development forum returns many instances of developers seeking a method of testing software GPIO interactions without the presence of the physical SBC and/or peripheral circuits. Suggestions to the solution vary, however they consistently include QEMU as a possible solution, and commonly also mention that QEMU does not support GPIO and therefore will only be useful for debugging the internal parts of the developer's project. Many threads continue to suggest that implementation of a virtual GPIO interface for QEMU should be possible but that it has not yet been done [2].

There is one similar implementation of a virtual GPIO interface for QEMU implemented for the Bifferboard SBC [3], however it is specific to emulating Intel 486 processors making it unusable for Raspberry Pi emulation. Additionally, the Bifferboard GPIO emulation uses a network socket for GPIO communication with the host PC, which does not provide the performance desired for this project. Some other QEMU SBC emulations provide a GPIO module for handling accesses to the memory-mapped I/O regions, however they do not provide interfaces for processes running on the host PC to access the GPIO interface [4].

Implementation of the virtual Raspberry Pi GPIO interface will therefore provide a capability strongly desired by the development community. Additionally, it will provide a platform for future implementation of Raspberry Pi functionality under QEMU.

Approach

DESIGN CONSTRAINTS

The two primary characteristics considered when choosing a method of implementation for this project were speed and code portability between the emulated platform and the real SBC.

Speed Requirement

Since the intent of this project is to emulate a hardware interface in which there is very little latency, it is important to minimize the latency induced by the virtual interface. In addition, the rate at which the state of the virtual interface can be modified must be maximized. As an example, if a project intended to use GPIO outputs as a high-speed serial interface, the virtual interface must be able to propagate information to and from the host fast enough for an accurate representation of the real interface. Speed should not be an issue for most host platforms, as most modern PCs have higher clock speeds than the ones in use in SBCs. However this does disqualify some simple methods of implementing the virtual interface. For example, a TCP/IP connection between the host and the emulated system is unsatisfactory due to network latency.

Code Portability Requirement

The second design driver, code portability, requires that the software under test not require modification to accommodate the virtual interface as this would diminish the effectiveness of the testing. Since the Raspberry Pi uses memory-mapped input/output to the GPIO pins, the virtual interface must be accessed by the software under test in the same manner, and the addresses used for real Raspberry Pi GPIO must also be used for the virtual interface.

BASELINE EMULATION

An important set of decisions for this project involve the baseline emulator configuration. A fundamental QEMU emulator configuration consists of 4 major elements – the CPU, the machine, the kernel, and the drive image(s) [5].

The core of the Raspberry Pi boards is a System-on-a-Chip (SoC) integrated circuit (IC). SoC's provide the fundamental components of a complete computer system (processor core,

memory, timing, peripheral interfaces, power management, graphics, etc.) on a single chip, and are commonly used in modern electronics since they reduce design and manufacturing cost, decrease hardware footprint, and diminish power consumption [6]. By contrast, microcontrollers are typically far simpler and require additional external components to operate as a full system. Powerful SoC's, like the Broadcom BCM2835 SoC used by the Raspberry Pi, may allow operation of the same operating systems in use by Desktop PCs. Emulation of the Raspberry Pi therefore entails a simulation of the various components that make up its SoC.

CPU Configuration

The CPU configuration defines the architecture of the emulated environment and its fundamental behavior in the same way that this selection does for real hardware. Raspberry Pi 1 models use an ARM 11 processor [7]. Raspberry Pi 2 and 3 models use newer ARM Cortex A7 and A53 models, respectively, the main difference from the ARM 11 series being number of cores and overall performance.

Conveniently, a QEMU CPU emulation exists for the ARM1176 processor series [5], which will be used for the baseline configuration for this project. Thus the emulation as a whole will be of a Raspberry Pi 1 model. Emulating Raspberry Pi 2 and 3 models would require a change of CPU configuration – these CPUs are available in the current QEMU package, however compatible machine configurations (as described below) are not. The selected configuration should be sufficient to test *most* software to be run on any of the Raspberry Pi models. The exception would be software written to use the few features specific to the newer ARM processor core families used on the Raspberry Pi 2 and 3 models.

Machine Configuration

The machine configuration defines the remaining emulated features of the SoC aside from the CPU, along with any other functional parts of the SBC such as external memory. Since the CPU is inseparable from the rest of the SoC, the machine selection must be compatible with the

CPU selection. Unfortunately, there is no existing machine configuration intended to match the Raspberry Pi hardware that makes a good baseline for this project.

In early 2016 QEMU added a partial Raspberry Pi machine for its version 2.6 release [8], which includes an emulation of the Broadcom BCM2835 SoC which is used on the Raspberry Pi 1. However, this emulation does not support the SoC's peripheral GPIO functionality, networking with the host computer, and a number of other device emulations. Due to these issues the version 2.6 Raspberry Pi machine was not used for this project.

A consistent recommendation by the Raspberry Pi development community is to use the Versatile Platform Baseboard (VersatilePB) machine available in QEMU [9]. The VersatilePB is a development system intended to support development using the ARM926EJ-S processor [10]. It provides many of the same features of the Raspberry Pi SoC, including GPIO, USB ports, ethernet, and support for SD cards (which the Raspberry Pi uses to load its operating system). In addition, it has an emulated VGA controller which allows QEMU to easily simulate a user display similar to that provided by the HDMI interface available on Raspberry Pi boards. Because of these compatibilities, the VersatilePB machine works well with the ARM1176 CPU emulation and with the Raspberry Pi software to the extent that developers use QEMU for testing software which does not use the Raspberry Pi hardware features.

Drive Image & Kernel Configuration

Raspbian is an operating system based on the Debian Linux distribution which is optimized to run on Raspberry Pi hardware [11]. The May 27, 2016 version of Raspbian (based on Debian version 8, also known as "Debian Jessie") was used for this project. The CPU and machine selections were close enough to the Raspberry Pi hardware that the Linux kernel included in the Raspbian distribution was used for the emulation without modification.

QEMU ARCHITECTURE

QEMU uses the virtualization architecture available on modern PC processors to manage translation between the emulated systems memory addresses and the host system memory

addresses [12]. This abstraction layer makes it impossible for QEMU to provide a way to determine the location of a certain guest memory address within the host memory. While the most obvious solution to this project would be to determine the host memory location corresponding to the emulated SBC's GPIO input/outputs, this is not possible. Instead, QEMU provides an emulated device object model in which devices may be assigned to memory regions and accesses to those memory regions can be detected and acted upon.

QDev Device Object Model

QEMU's device object model architecture, known as QDev [13], is made up of 2 types of entities – devices and busses – which are represented by device states and bus states, respectively. Devices may have extended properties while busses may not; the sole purpose of a bus is to connect devices. The main bus in any QEMU machine is the System Bus (or SysBus). A machine configuration includes the set of busses and devices which make up the machine, minimally consisting of the SysBus. Lower-tiered busses may be added if the machine configuration being emulated requires it. For example, the SysBus may be connected to a PCI controller device, which controls a PCI bus, to which other devices are connected.

Device Memory

Memory in QEMU is separate from QDev, but QDev devices may have their states assigned to memory ranges [14]. For example, a UART device is not only attached to a bus in QDev, its state is also mapped to a memory address. Part of this UART device's state would be a transmit shift register so that when processes write to the memory location corresponding to the device's shift register, the code within the device definition would shift out the data to emulate what a real UART would do. Once the device has been initialized and assigned to a bus, reads and writes to the device's state memory region are detected by the QDev framework and are passed on to the device's functional code for any further action.

EXTERNAL DEVICE EMULATION

Once the virtual interface is in place, there should be a way for the host to emulate the hardware which is connected to the GPIO pin(s). This could be done in numerous ways such as a visual depiction of peripherals or an automated script. As an example of one such emulation, this project will implement LED and pushbutton widgets to be used in the popular user interface creator, Qt Designer, distributed as part of QtCreator development environment [15].

Implementation

RASPBERRY PI GPIO DEVICE

The primary addition made to QEMU to allow operation of the virtual interface is the Raspberry PI GPIO device, RPI_GPIO, which extends the QDev device class. RPI_GPIO's device state structure, RPI_GPIO_State, is based on the full 54-channel GPIO capability of the BCM2835 SoC which powers the Raspberry Pi 1 board [16]. While the Raspberry Pi only extends 8 of the54 GPIO channels to the pin header for general I/O use [17], inclusion of all 54 channels allows future expansion of the device's emulated capabilities to include access to other functions implemented by the Raspberry Pi such as reading SD-Cards, interfacing with a JTAG loader, etc.

Device State

The device state, RPI_GPIO_State, includes all 29 registers implemented by the BCM2835 GPIO module. Not all registers are currently used by the emulated device. For example, the edge detect interrupts have not been implemented in the device – this is left as a simple future addition.

Refer to the following table for the function of each register included in the device state [16]:

Register Name	Purpose	Implemented Use
GPFSEL0	Function Select Pins 0-9	Selects input or output
GPFSEL1	Function Select Pins 10-19	Selects input or output
GPFSEL2	Function Select Pins 20-29	Selects input or output
GPFSEL3	Function Select Pins 30-39	Selects input or output
GPFSEL4	Function Select Pins 40-49	Selects input or output
GPFSEL5	Function Select Pins 50-53	Selects input or output
GPSET0	Output Set Pins 0-31	Sets output pin level to 1
GPSET1	Output Set Pins 32-53	Sets output pin level to 1
GPCLR0	Output Clear Pins 0-31	Sets output pin level to 0
GPCLR1	Output Clear Pins 32-53	Sets output pin level to 0
GPLEV0	Pin Level (Read Only) 0-31	Reflects input pin level
GPLEV1	Pin Level (Read Only) 32-53	Reflects input pin level
GPEDS0	Event Detect Pins 0-31	Unused but available
GPEDS1	Event Detect Pins 32-53	Unused but available
GPREN0	Rising Edge Detect Enable Pins 0-31	Unused but available
GPREN1	Rising Edge Detect Enable Pins 32-53	Unused but available
GPFEN0	Falling Edge Detect Enable Pins 0-31	Unused but available
GPFEN1	Falling Edge Detect Enable Pins 32-53	Unused but available
GPHEN0	High Level Detect Enable Bits 0-31	Unused but available
GPHEN1	High Level Detect Enable Bits 32-53	Unused but available
GPLEN0	Low Level Detect Enable Bits 0-31	Unused but available
GPLEN1	Low Level Detect Enable Bits 32-53	Unused but available
GPAREN0	Async Rising Edge Detect Enable Pins 0-31	Unused but available
GPAREN1	Async Rising Edge Detect Enable Pins 32-53	Unused but available
GPAFEN0	Async Falling Edge Detect Enable Pins 0-31	Unused but available
GPAFEN1	Async Falling Edge Detect Enable Pins 32-53	Unused but available
GPPUD	Pull-Up/Pull-Down All Pins	Unused but available
GPPUDCLK0	Pull-Up/Pull-Down Clock Pins 0-31	Unused but available
GPPUDCLK1	Pull-Up/Pull-Down Clock Pins 32-53	Unused but available

Table 1:   BCM2835 GPIO Registers

In addition to the BCM2835 GPIO registers several calculated fields have been included in the RPI_GPIO_State structure. The *OUTSTATE0* and *OUTSTATE1* fields contain the output states for each pin which are set high or low according to the *GPSETx* and *GPCLRx* registers (a function of the BCM2835 SoC when using real hardware; *x* represents the number of the corresponding register). A pointer to a *shared_gpio_state* structure is also included in RPI_GPIO_State, which contains a subset of the RPI_GPIO_State fields which will be shared with the host. The shared state includes the 6 function registers (*GPFSELx*), the input level registers (*GPLVLx*), and the output states (*OUTSTATEx*).

Lastly, the reserved regions within the BCM2835 GPIO register range are also included in the device state so that the emulated device's memory region matches all offsets of the real BCM2835.

Appendix A contains the source code of the RPI_GPIO QEMU device.

Memory Access Functions

QDev provides read() and write() callback function slots which can be implemented in the device and are called whenever the device's assigned memory region is read or written to, respectively. In the RPI_GPIO device, a write access causes an update of the RPI_GPIO_State fields according to a passed-in state object. In addition, the output levels are adjusted according to the appropriate registers, and the *shared_gpio_state* fields are updated to match the device state. A read request includes an offset parameter which is relative to the start of the device state memory, and the appropriate register is returned. A read request also causes the *shared_gpio_state GPLEVx* register bits to be copied into the device state for all inputs.

SHARED MEMORY INITIALIZATION

An important feature of the RPI_GPIO device is the initialization of a shared memory region that can be accessed by the host. The *shmget()* and *shmat()* system functions are used to allocate the shared memory region, attach it to the QEMU address space, and then the shared state pointer within the RPI_GPIO_State is set to the shared region.

VERSATILEPB MACHINE MODIFICATION

The only modification required to the existing VersatilePB machine configuration was to attach an instance of the new RPI_GPIO device to the VersatilePB's SysBus, and assign it to the BCM2835's GPIO memory region starting at 0x20200000. This address range was unused within the VersatilePB machine's memory. The modification was accomplished with a single call to QDev's *sysbus_create_simple* function with the base memory address as one of the parameters.

To use the VersatilePB machine to emulate Raspberry Pi 2 or 3 devices, the starting address of the RPI_GPIO device's memory region would need to be updated to match the correct address for that hardware version.

EXTERNAL HARDWARE EMULATION

Once the *shared_gpio_state* is allocated as shared memory, other processes on the host PC are free to access it. This can be used to develop PC-side tools for testing the emulated Raspberry Pi's GPIO functions. As a demonstration of this ability, two Qt Designer widgets were created – an LED and a button. The widgets can be used in Qt Designer to build GUIs which represent the real hardware being emulated. Figure 1, below, shows the QT Designer window with a form under development. On the form is a layout of the GPIO-connected LEDs and buttons. To the right, included in the selected LED's property settings, are the GPIO Pin and Refresh Rate settings.
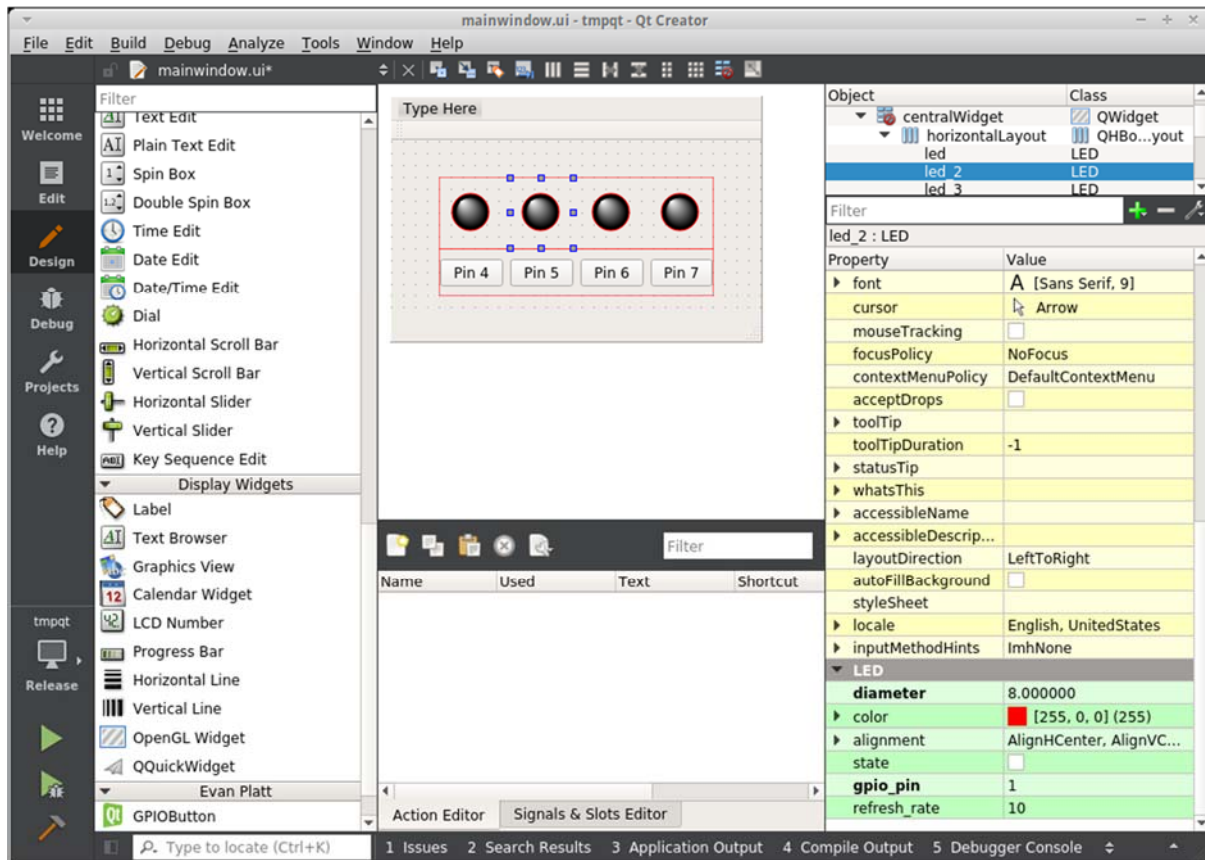
Figure 1: QT Designer GUI built using GPIO LED and button widgets

The LED widget has two custom parameters – the GPIO Pin to which it is connected and the refresh rate in milliseconds. Upon initialization of the LED widget a continuous periodic timer is started, timing out every time the refresh rate has elapsed and causing the LED's on/off state to be updated according to the GPIO output state in shared memory.

The button widget also has two custom parameters – the GPIO pin to which it is connected and the push duration in milliseconds. When the button is pressed, a 1 is sent to the GPLEV register bit in shared memory that corresponds to the GPIO pin setting, and a 0 is written once the specified push duration is reached.

Appendix B contains the source code of the two widgets.

TEST SOFTWARE

The last piece of software required to test the GPIO functionality was a program to run within the QEMU-emulated Raspberry Pi. This represents the "software under test" that would be loaded to the real Raspberry Pi SBC assuming the test with the virtual GPIO interface showed that the software met the developer's needs.

There are libraries available in several different programming languages which simplify the method in which programs access the Raspberry Pi GPIO pins. The most popular is wiringPi, a C library that provides both a command line interface to directly access the GPIO pins one command at a time, and a library to be included in other programs [18]. WiringPi had to be modified to run under QEMU since it uses the data in /proc/cpuinfo to determine what revision of Raspberry Pi is being used. A new function was added to check if a QEMU emulation is being used, and if so, a configuration file required to be in the user's home folder is used to set the board id and revision parameters needed to determine the GPIO base address. For the purposes of testing the QEMU CPU and machine configurations in use (which emulate a Raspberry Pi 1 SBC), the test PC's configuration file was set to specify a Raspberry Pi 1 wiringPi configuration. Appendix C contains the modifications made to the wiringPi library.

A simple test program was written using the methods of the wiringPi library as modified. It configures GPIO channels 0-3 as outputs and 4-7 as inputs. In a continuous loop, the channel 0-3 outputs are set according to the channel 4-7 inputs (channel 4's input state appears on output channel 0, channel 5's input state appears on output channel 1, etc.). Appendix D contains the source code of the test program.

Results

INPUT/OUTPUT TEST

The first test run against the virtual GPIO interface used the test Raspberry Pi program along with a Qt program using the custom widgets to exercise each GPIO pin. A Qt form was prepared using 4 LED widgets and 4 button widgets to correspond with the test program inputs and outputs. Figure 2, below, shows the test form with 1 row of LEDs (set to GPIO pins 0-3) and 1 row of buttons (set to GPIO pins 4-7).
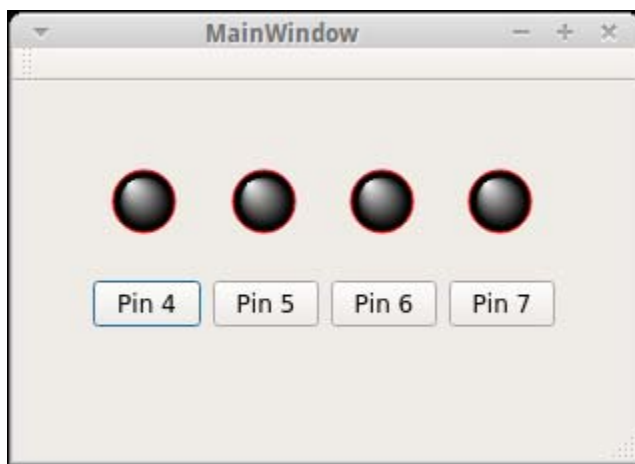

Figure 2:   Test form with all GPIO outputs low

Each GPIO pin was tested by pressing one of the 4 buttons and verifying that the corresponding LED turned on for the button's configured push duration. Figure 3 shows the form just after the "Pin 4" button was clicked (GPIO input 5 causing GPIO output 1 to go high):
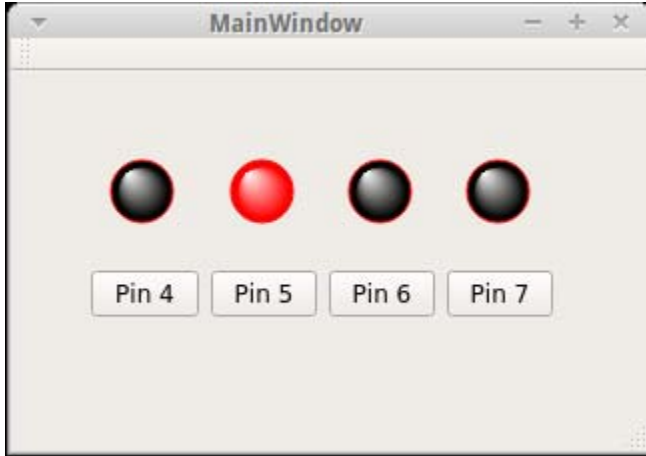
Figure 3:   Test form with GPIO output 2 high in response to GPIO input 5

IMPLEMENTATION REQUIREMENTS VERIFICATION

Speed Requirement

Since there is no enforced time consistency between the emulation and the host, it is not possible to obtain a read or write throughput measurement for the virtual GPIO interface. However it is possible to measure the time between the manipulation of an input by the host and an associated output from the emulation. A test was performed using a modified version of the test program in Appendix D running on the Raspberry Pi, and a test script on the host PC. The script simply set an input pin and waited for the associated output pin to go high. The time difference from when the input pin was set to when the output pin went high was considered to be the round-trip delay. The test program modification reduced the loop to only write the input pin 4 value onto output pin 0 in order to avoid extra latency induced by writing/reading the other 3 sets of pins. The average delay of 1000 trials was used to calculate a delay of 1.2μs for the round trip. Figure 4 shows the round-trip time for the 1000 trials to illustrate the variability of throughput.

16

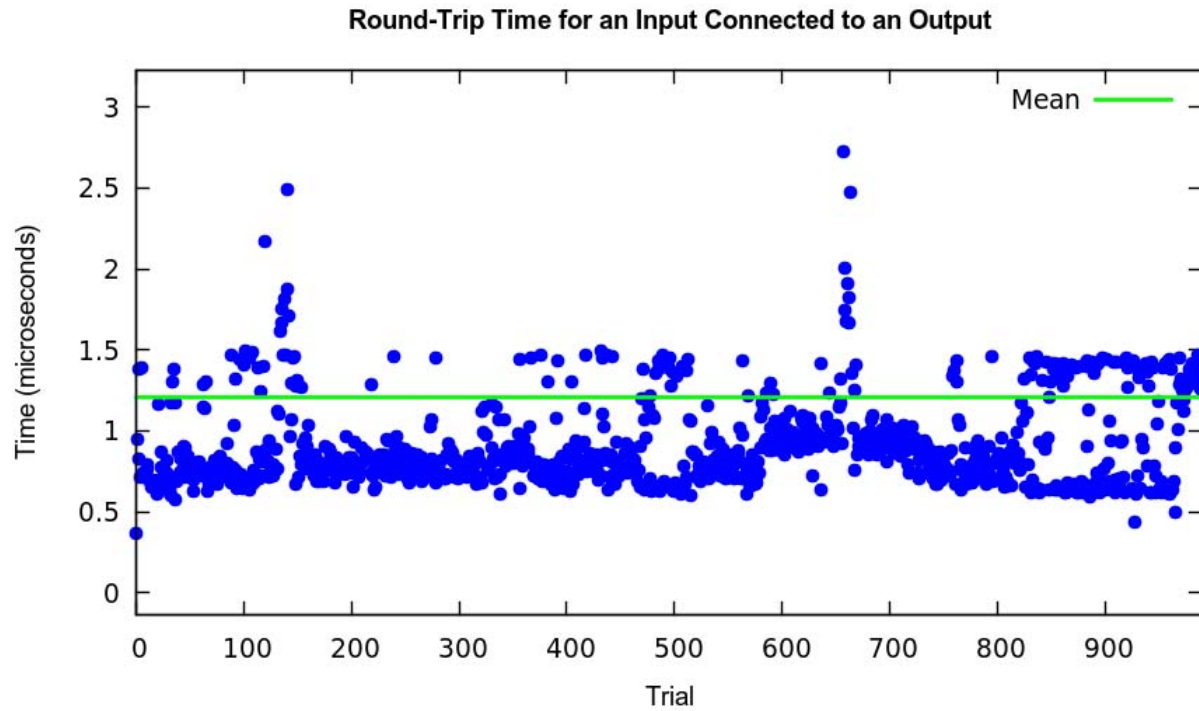**Round-Trip Time for an Input Connected to an Output**

Figure 4: Round-Trip Time for an Input Connected to an Output

In the plot shown in Figure 4, nine outliers have been removed so that the rest of the data could be better represented. These outliers were distributed between 10µs to 80µs round trip times. In order to explain these outliers, another test was performed on the Raspberry Pi emulation (without virtual GPIO interactions) to measure the time required for 1000 simple writes to memory. This test showed a similar frequency and magnitude of outliers, indicating that the occasional delays were not caused by the virtual GPIO port but are inherent to the QEMU memory system. Figure 5 shows both the GPIO round-trip times (now including the outliers) along with the simple memory write times. Note that the data has been normalized with respect to the y-axis – this is due to the times recorded for the simple write access (mean 24,634ns) being much higher than the GPIO round-trip times (mean 1208ns) due to the emulated system's time running much faster than the PC's "real time".
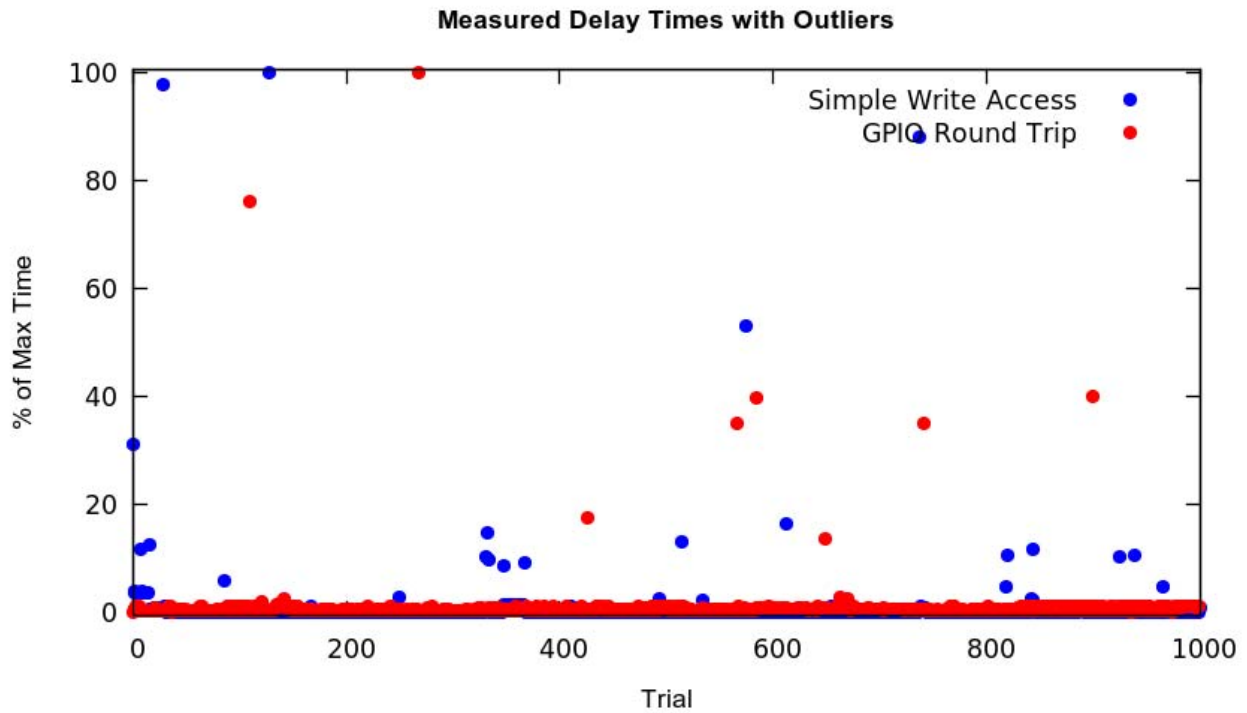
Figure 5: Delay times of GPIO round trips and simple write accesses, including outliers

A second test was run to show that QEMU's QDev device schema can handle a high rate of GPIO writes. For this test a program was written on the Raspberry Pi to cycle one pin, high then low, 1000 times, with no delay. A debug variable was added to the RPI_GPIO device code to keep track of write updates. Running the test program consistently resulted in 2000 increments of the write counter (one for each low-to-high transition and one for each high-to-low). This shows that the update rate of the virtual GPIO interface is limited only by the CPU frequency of the emulation, and therefore the host PC's CPU frequency. QEMU does not provide CPU frequency modification. If a particular test needs accurate timing similar to the Raspberry Pi processor frequency then the virtual interface may not be a sufficient representation of the real hardware for testing purposes.

18

Code Portability Requirement

Since the QEMU RPI_GPIO device is assigned the same base address as the memory-mapped IO region of a real Raspberry Pi, software changes to the way GPIO is accessed are not required to use the emulation.

Although no modifications were required to use the test program either on the emulated Raspberry Pi or a real SBC, a change was required to the wiringPi library to allow it to support both real SBCs and the QEMU emulation. Since wiringPi is a software abstraction, and the modification did not affect the way in which wiringPi interacts with the SBC's GPIO features, this modification does not constitute a failure to fulfill the code portability constraint. The version of wiringPi produced as part of this project will work on both real and emulated Raspberry Pi computers.

Conclusions

This study created a virtual GPIO interface for PC-emulated Raspberry Pi SBC and showed how this interface can be used for testing GPIO interactions of software running on the SBC. While only basic logic-level functionality of the real GPIO interface was implemented, the emulated GPIO device provides a framework for easily adding other capabilities such as event-driven interrupts and pull-up/pull-down emulation. In addition, since the RPI_GPIO device offers the full register compliment of the BCM2835 SoC, it is ready to be integrated with more complex communication modules, such as Universal Asynchronous Receiver/Transmitter (UART) and Serial Peripheral Interface (SPI), which are also offered by the Raspberry Pi.

The LED and pushbutton widgets created for use in the Qt Designer user interface platform are only examples of host-side endpoints for the virtual GPIO interface. There are many possibilities for what a developer would want to implement on the host side depending on what GPIO functionality their project employs. The host-side peripheral could also be represented by a script, some other graphical representation, or even a peripheral driver connecting the emulated GPIO to real hardware connected to the host's PC.

The best existing solution for Raspberry Pi emulation under QEMU consists of the Raspbian operating system running on an emulation of a different hardware platform. This configuration may be sufficient for testing some embedded software that doesn't require external peripherals. However when GPIO becomes involved, there is currently no community standard for how to interactively debug without running on the actual SBC. The virtual GPIO interface provides this capability.

With hope that the embedded software community will expand upon the capabilities implemented thus far, the source code implemented during this project has been made publicly available on GitHub at http://github.com/evplatt/mse_report.

Appendices

APPENDIX A: SOURCE CODE FOR QEMU DEVICE

```c
/*
 * BCM2835 General Purpose IO Module for Raspberry Pi
 * Also available at http://github.com/evplatt/mse_report/qemu
 */

#include "qemu/osdep.h"
#include "hw/sysbus.h"
#include <sys/shm.h>
#include <errno.h>
#include <string.h>

/* Macros to enable debug messages */
#ifdef DEBUG_RPI_GPIO
#define DPRINTF(fmt, ...) \
do { printf("rpi_gpio: " fmt , ## __VA_ARGS__); } while (0)
#define BADF(fmt, ...) \
do { fprintf(stderr, "rpi_gpio: error: " fmt , ## __VA_ARGS__); exit(1);}
while (0)
#else
#define DPRINTF(fmt, ...) do {} while(0)
#define BADF(fmt, ...) \
do { fprintf(stderr, "rpi_gpio: error: " fmt , ## __VA_ARGS__);} while (0)
#endif

#define TYPE_RPI_GPIO "rpi_gpio"
#define RPI_GPIO(obj) OBJECT_CHECK(RPI_GPIO_State, (obj), TYPE_RPI_GPIO)

/* shared_gpio_state includes the registers to be shared with the host */
typedef struct shared_gpio_state {

  uint32_t GPFSEL0;    /* Function Select Pins 0-9   */
  uint32_t GPFSEL1;    /* Function Select Pins 10-19 */
  uint32_t GPFSEL2;    /* Function Select Pins 20-29 */
  uint32_t GPFSEL3;    /* Function Select Pins 30-39 */
  uint32_t GPFSEL4;    /* Function Select Pins 40-49 */
  uint32_t GPFSEL5;    /* Function Select Pins 50-53 */
  uint32_t GPLEV0;     /* Input level register for pins  0-31 */
  uint32_t GPLEV1;     /* Input level register for pins 32-53 */
  uint32_t OUTSTATE0;  /* Derived output state for pins  0-31 */
  uint32_t OUTSTATE1;  /* Derived output state for pins 32-53 */

} shared_gpio_state;

/* RPI_GPIO_State represents the device and its memory structure. */
/* Refer to section 6.1 of the Broadcom BCM2835 ARM Peripherials Guide */
typedef struct RPI_GPIO_State {
    SysBusDevice parent_obj;
    MemoryRegion iomem;
    uint32_t GPFSEL0;   /* 0x00 Function Select Pins 0-9   */
    uint32_t GPFSEL1;   /* 0x04 Function Select Pins 10-19 */
    uint32_t GPFSEL2;   /* 0x08 Function Select Pins 20-29 */
```

```c
    uint32_t GPFSEL3;    /* 0x0c Function Select Pins 30-39 */
    uint32_t GPFSEL4;    /* 0x10 Function Select Pins 40-49 */
    uint32_t GPFSEL5;    /* 0x14 Function Select Pins 50-53 */
    uint32_t res0;       /* 0x18 */
    uint32_t GPSET0;     /* 0x1c Output Set Pins  0-31 */
    uint32_t GPSET1;     /* 0x20 Output Set Pins 32-53 */
    uint32_t res1;       /* 0x24 */
    uint32_t GPCLR0;     /* 0x28 Output Clear Pins  0-31 */
    uint32_t GPCLR1;     /* 0x2c Output Clear Pins 32-53 */
    uint32_t res2;       /* 0x30 */
    uint32_t GPLEV0;     /* 0x34 Pin Level (Read Only)  0-31 */
    uint32_t GPLEV1;     /* 0x38 Pin Level (Read Only) 22-53 */
    uint32_t res3;       /* 0x3c */
    uint32_t GPEDS0;     /* 0x40 Event Detect Pins  0-31 */
    uint32_t GPEDS1;     /* 0x44 Event Detect Pins 32-53 */
    uint32_t res4;       /* 0x48 */
    uint32_t GPREN0;     /* 0x4c Rising Edge Detect Enable Pins  0-31 */
    uint32_t GPREN1;     /* 0x50 Rising Edge Detect Enable Pins 32-53 */
    uint32_t res5;       /* 0x54 */
    uint32_t GPFEN0;     /* 0x58 Falling Edge Detect Enable Pins  0-31 */
    uint32_t GPFEN1;     /* 0x5c Falling Edge Detect Enable Pins 32-53 */
    uint32_t res6;       /* 0x60 */
    uint32_t GPHEN0;     /* 0x64 High Level Detect Enable Bits  0-31 */
    uint32_t GPHEN1;     /* 0x68 High Level Detect Enable Bits 32-53 */
    uint32_t res7;       /* 0x6c */
    uint32_t GPLEN0;     /* 0x70 Low Level Detect Enable Bits  0-31 */
    uint32_t GPLEN1;     /* 0x74 Low Level Detect Enable Bits 32-53 */
    uint32_t res8;       /* 0x78 */
    uint32_t GPAREN0;    /* 0x7c Async Rising Edge Detect Enable Pins  0-31 */
    uint32_t GPAREN1;    /* 0x80 Async Rising Edge Detect Enable Pins 32-53 */
    uint32_t res9;       /* 0x84 */
    uint32_t GPAFEN0;    /* 0x88 Async Falling Edge Detect Ena Pins  0-31 */
    uint32_t GPAFEN1;    /* 0x8c Async Falling Edge Detect Ena Pins 32-53 */
    uint32_t res10;      /* 0x90 */
    uint32_t GPPUD;      /* 0x94 Pull-Up/Pull-Down All Pins */
    uint32_t GPPUDCLK0;  /* 0x98 Pull-Up/Pull-Down Clock Pins  0-31 */
    uint32_t GPPUDCLK1;  /* 0x9c Pull-Up/Pull-Down Clock Pins 32-53 */
    uint32_t res11;      /* 0xa0 */
    uint32_t test;       /* 0xb0 */
    uint32_t OUTSTATE0;  /* Derived output state for pins  0-31 */
    uint32_t OUTSTATE1;  /* Derived output state for pins 32-53 */
    qemu_irq out[54];
    shared_gpio_state *shm;  /* pointer to shared state */
    const unsigned char *id;
} RPI_GPIO_State;

/* Device desciption required by QDev */
static const VMStateDescription vmstate_rpi_gpio = {
    .name = "rpi_gpio",
    .version_id = 0,
    .minimum_version_id = 0,
    .fields = (VMStateField[]) {
      VMSTATE_UINT32(GPFSEL0, RPI_GPIO_State),
      VMSTATE_UINT32(GPFSEL1, RPI_GPIO_State),
      VMSTATE_UINT32(GPFSEL2, RPI_GPIO_State),
      VMSTATE_UINT32(GPFSEL3, RPI_GPIO_State),
      VMSTATE_UINT32(GPFSEL4, RPI_GPIO_State),
```

```
        VMSTATE_UINT32(GPFSEL5, RPI_GPIO_State),
        VMSTATE_UINT32(GPSET0, RPI_GPIO_State),
        VMSTATE_UINT32(GPSET1, RPI_GPIO_State),
        VMSTATE_UINT32(GPCLR0, RPI_GPIO_State),
        VMSTATE_UINT32(GPCLR1, RPI_GPIO_State),
        VMSTATE_UINT32(GPLEV0, RPI_GPIO_State),
        VMSTATE_UINT32(GPLEV1, RPI_GPIO_State),
        VMSTATE_UINT32(GPEDS0, RPI_GPIO_State),
        VMSTATE_UINT32(GPEDS1, RPI_GPIO_State),
        VMSTATE_UINT32(GPREN0, RPI_GPIO_State),
        VMSTATE_UINT32(GPREN1, RPI_GPIO_State),
        VMSTATE_UINT32(GPFEN0, RPI_GPIO_State),
        VMSTATE_UINT32(GPFEN1, RPI_GPIO_State),
        VMSTATE_UINT32(GPHEN0, RPI_GPIO_State),
        VMSTATE_UINT32(GPHEN1, RPI_GPIO_State),
        VMSTATE_UINT32(GPLEN0, RPI_GPIO_State),
        VMSTATE_UINT32(GPLEN1, RPI_GPIO_State),
        VMSTATE_UINT32(GPAREN0, RPI_GPIO_State),
        VMSTATE_UINT32(GPAREN1, RPI_GPIO_State),
        VMSTATE_UINT32(GPAFEN0, RPI_GPIO_State),
        VMSTATE_UINT32(GPAFEN1, RPI_GPIO_State),
        VMSTATE_UINT32(GPPUD, RPI_GPIO_State),
        VMSTATE_UINT32(GPPUDCLK0, RPI_GPIO_State),
        VMSTATE_UINT32(GPPUDCLK1, RPI_GPIO_State),
        VMSTATE_END_OF_LIST()
    }
};

/* Given a device state and pin number, returns the current pin function
selection */
static uint32_t rpi_get_pin_function(RPI_GPIO_State *s, int pin){

  /* GPFSELx registers have 3 bits per pin, 10 pins per register */
  if (pin<10)           return ((s->GPFSEL0 >> (3*pin))      & 0x7);
  if (pin>=10 && pin<20) return ((s->GPFSEL1 >> (3*(pin-10))) & 0x7);
  if (pin>=20 && pin<30) return ((s->GPFSEL2 >> (3*(pin-20))) & 0x7);
  if (pin>=30 && pin<40) return ((s->GPFSEL3 >> (3*(pin-30))) & 0x7);
  if (pin>=40 && pin<50) return ((s->GPFSEL4 >> (3*(pin-40))) & 0x7);
  return ((s->GPFSEL5 >> (3*(pin-50))) & 0x7);


}

/* Write Update function called after a write detection performs the
following tasks:
      1.  Calculates OUTSTATE fields according to GPSETx and GPCLRx registers
      2.  Updates the shared_gpio_state
*/
static void rpi_gpio_update(RPI_GPIO_State *s)
{
  int i;
  uint32_t mask;

  for (i=0; i<32; i++){
    mask = (1 << i);
    if (rpi_get_pin_function(s,i) == 1){   /* make sure its an output */
      if (s->GPSET0 & mask){               /* set flag is set */
        s->OUTSTATE0 |= mask;              /* set the state bit */
```

```c
            s->GPSET0 &= ~mask;                    /* clear the set bit */
        }
        else if (s->GPCLR0 & mask){                /* clear flag is set */
            s->OUTSTATE0 &= ~mask;                 /* set the state bit */
            s->GPCLR0 &= ~mask;                    /* clear the set bit */
        }
      }
    }
  }
  for (i=32; i<54; i++){
    mask = (1 << (i-32));
    if (rpi_get_pin_function(s,i) == 1){    /* make sure its an output */
      if (s->GPSET1 & mask){                       /* set flag is set */
        s->OUTSTATE1 |= mask;                      /* set the state bit */
        s->GPSET1 &= ~mask;                        /* clear the set bit */
      }
      else if (s->GPCLR1 & mask){                  /* clear flag is set */
        s->OUTSTATE1 &= ~mask;                     /* set the state bit */
        s->GPCLR1 &= ~mask;                        /* clear the set bit */
      }
    }
  }

  s->shm->GPFSEL0 = s->GPFSEL0;
  s->shm->GPFSEL1 = s->GPFSEL1;
  s->shm->GPFSEL2 = s->GPFSEL2;
  s->shm->GPFSEL3 = s->GPFSEL3;
  s->shm->GPFSEL4 = s->GPFSEL4;
  s->shm->GPFSEL5 = s->GPFSEL5;
  s->shm->OUTSTATE0 = s->OUTSTATE0;
  s->shm->OUTSTATE1 = s->OUTSTATE1;

}

/* Read Update function called after a read detection is used to
   copy the GPLEVx registers (which may have been updated by the
   emulation host) to the device state
*/
static void rpi_gpio_update_from_shared(RPI_GPIO_State *s)
{

  int i, mask;

  for (i=0; i<32; i++){
    mask = (1 << i);
    if (rpi_get_pin_function(s,i) == 0){ /* only check input pins */
      if ((mask & s->shm->GPLEV0) > 0) s->GPLEV0 |= mask;
      else s->GPLEV0 &= ~mask;
    }
  }
  for (i=32; i<54; i++){
    mask = (1 << i);
    if (rpi_get_pin_function(s,i) == 0){ /* only check input pins */
      if ((mask & s->shm->GPLEV1) > 0) s->GPLEV1 |= mask;
      else s->GPLEV1 &= ~mask;
    }
  }
```

```c
}


/* Called by QDev upon read detection
   Returns the device state field corresponding to the read address
*/
static uint64_t rpi_gpio_read(void *opaque, hwaddr offset,
                              unsigned size)
{
    RPI_GPIO_State *s = (RPI_GPIO_State *)opaque;

    rpi_gpio_update_from_shared(s);  /* First update any input pins
                                        from the shared_gpio_state */

    DPRINTF("rpi_gpio_debug:  rpi_gpio_read - offset=%02x, size=%d\n",
            (int)offset, size);

    switch (offset) {
      case 0x00:
          return s->GPFSEL0;
      case 0x04:
          return s->GPFSEL1;
      case 0x08:
          return s->GPFSEL2;
      case 0x0c:
          return s->GPFSEL3;
      case 0x10:
          return s->GPFSEL4;
      case 0x14:
          return s->GPFSEL5;
      case 0x34:
          return s->GPLEV0;
      case 0x38:
          return s->GPLEV1;
      case 0x40:
          return s->GPEDS0;
      case 0x44:
          return s->GPEDS1;
      case 0x4c:
          return s->GPREN0;
      case 0x50:
          return s->GPREN1;
      case 0x58:
          return s->GPFEN0;
      case 0x5c:
          return s->GPFEN1;
      case 0x64:
          return s->GPHEN0;
      case 0x68:
          return s->GPHEN1;
      case 0x70:
          return s->GPLEN0;
      case 0x74:
          return s->GPLEN1;
      case 0x7c:
          return s->GPAREN0;
      case 0x80:
```

```
            return s->GPAREN1;
        case 0x88:
            return s->GPAFEN0;
        case 0x8c:
            return s->GPAFEN1;
        case 0x94:
            return s->GPPUD;
        case 0x98:
            return s->GPPUDCLK0;
        case 0x9c:
            return s->GPPUDCLK1;
        case 0x1c: /* GPSET0 (Write-Only) */
        case 0x20: /* GPSET1 (Write-Only) */
        case 0x28: /* GPCLR0 (Write-Only) */
        case 0x2c: /* GPCLR1 (Write-Only) */
        case 0x18: /* res0 */
        case 0x24: /* res1 */
        case 0x30: /* res2 */
        case 0x3c: /* res3 */
        case 0x48: /* res4 */
        case 0x54: /* res5 */
        case 0x60: /* res6 */
        case 0x6c: /* res7 */
        case 0x78: /* res8 */
        case 0x84: /* res9 */
        case 0x90: /* res10 */
        case 0xa0: /* res11 */
        case 0xb0: /* test */
            goto err_out;
        default:
            break;
    }
err_out:
    qemu_log_mask(LOG_GUEST_ERROR,
                  "rpi_gpio_read: Bad offset %x\n", (int)offset);
    return 0;
}

/* Called by QDev upon write detection
   Updates the device state according to the manipulated memory state
*/
static void rpi_gpio_write(void *opaque, hwaddr offset,
                       uint64_t value, unsigned size)
{
    RPI_GPIO_State *s = (RPI_GPIO_State *)opaque;

    DPRINTF("rpi_gpio_write - offset=%02x, size=%d, value=%d\n",
                    (int)offset, size, (uint32_t)value);

    switch (offset) {
      case 0x00:
          s->GPFSEL0 = (value & 0xffffffff);
          break;
      case 0x04:
          s->GPFSEL1 = (value & 0xffffffff);
          break;
      case 0x08:
```

```c
        s->GPFSEL2 = (value & 0xffffffff);
        break;
    case 0x0c:
        s->GPFSEL3 = (value & 0xffffffff);
        break;
    case 0x10:
        s->GPFSEL4 = (value & 0xffffffff);
        break;
    case 0x14:
        s->GPFSEL5 = (value & 0xffffffff);
        break;
    case 0x1c:
        s->GPSET0 = (value & 0xffffffff);
        break;
    case 0x20:
        s->GPSET1 = (value & 0xffffffff);
        break;
    case 0x28:
        s->GPCLR0 = (value & 0xffffffff);
        break;
    case 0x2c:
        s->GPCLR1 = (value & 0xffffffff);
        break;
    case 0x40:
        s->GPEDS0 = (value & 0xffffffff);
        break;
    case 0x44:
        s->GPEDS1 = (value & 0xffffffff);
        break;
    case 0x4c:
        s->GPREN0 = (value & 0xffffffff);
        break;
    case 0x50:
        s->GPREN1 = (value & 0xffffffff);
        break;
    case 0x58:
        s->GPFEN0 = (value & 0xffffffff);
        break;
    case 0x5c:
        s->GPFEN1 = (value & 0xffffffff);
        break;
    case 0x64:
        s->GPHEN0 = (value & 0xffffffff);
        break;
    case 0x68:
        s->GPHEN1 = (value & 0xffffffff);
        break;
    case 0x70:
        s->GPLEN0 = (value & 0xffffffff);
        break;
    case 0x74:
        s->GPLEN1 = (value & 0xffffffff);
        break;
    case 0x7c:
        s->GPAREN0 = (value & 0xffffffff);
        break;
    case 0x80:
```

```c
            s->GPAREN1 = (value & 0xffffffff);
            break;
        case 0x88:
            s->GPAFEN0 = (value & 0xffffffff);
            break;
        case 0x8c:
            s->GPAFEN1 = (value & 0xffffffff);
            break;
        case 0x94:
            s->GPPUD = (value & 0xffffffff);
            break;
        case 0x98:
            s->GPPUDCLK0 = (value & 0xffffffff);
            break;
        case 0x9c:
            s->GPPUDCLK1 = (value & 0xffffffff);
            break;
        case 0x34: /* GPLEV0 (Read-Only) */
        case 0x38: /* GPLEV1 (Read-Only) */
        case 0x18: /* res0 */
        case 0x24: /* res1 */
        case 0x30: /* res2 */
        case 0x3c: /* res3 */
        case 0x48: /* res4 */
        case 0x54: /* res5 */
        case 0x60: /* res6 */
        case 0x6c: /* res7 */
        case 0x78: /* res8 */
        case 0x84: /* res9 */
        case 0x90: /* res10 */
        case 0xa0: /* res11 */
        case 0xb0: /* test */
        default:
            goto err_out;
    }
    rpi_gpio_update(s); /* Set output levels and update shared_gpio_state */
    return;
err_out:
    qemu_log_mask(LOG_GUEST_ERROR,
                "rpi_gpio_write: Bad offset %x\n", (int)offset);
}

static void rpi_gpio_reset(DeviceState *dev)
{
    RPI_GPIO_State *s = RPI_GPIO(dev);

    s->GPFSEL0   = 0;
    s->GPFSEL1   = 0;
    s->GPFSEL2   = 0;
    s->GPFSEL3   = 0;
    s->GPFSEL4   = 0;
    s->GPFSEL5   = 0;
    s->GPSET0    = 0;
    s->GPSET1    = 0;
    s->GPCLR0    = 0;
    s->GPCLR1    = 0;
    s->GPLEV0    = 0;
```

```c
      s->GPLEV1    = 0;
      s->GPEDS0    = 0;
      s->GPEDS1    = 0;
      s->GPREN0    = 0;
      s->GPREN1    = 0;
      s->GPFEN0    = 0;
      s->GPFEN1    = 0;
      s->GPHEN0    = 0;
      s->GPHEN1    = 0;
      s->GPLEN0    = 0;
      s->GPLEN1    = 0;
      s->GPAREN0   = 0;
      s->GPAREN1   = 0;
      s->GPAFEN0   = 0;
      s->GPAFEN1   = 0;
      s->GPPUD     = 0;
      s->GPPUDCLK0 = 0;
      s->GPPUDCLK1 = 0;

}

/* QDev-required set function */
static void rpi_gpio_set(void * opaque, int line, int level)
{
  RPI_GPIO_State *s = (RPI_GPIO_State *)opaque;

  if (rpi_get_pin_function(s,line) == 0){

    /* We have an input; Set the level */
    if (line<32){
      uint32_t mask = 1 << line;
      s->GPLEV0 &= ~mask;
      if (level) s->GPLEV0 |= mask;
    }
    else{
      uint32_t mask = 1 << (line-32);
      s->GPLEV1 &= ~mask;
      if (level) s->GPLEV1 |= mask;
    }
  }
}

/* Tie the read/write functions above to QDev */
static const MemoryRegionOps rpi_gpio_ops = {
    .read = rpi_gpio_read,
    .write = rpi_gpio_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
};

/* Use the POSIX shm functions to create a shared memory region
   and map it into QEMU's memory.  Return the pointer.
*/
static shared_gpio_state *get_shared_ptr(void);
static shared_gpio_state *get_shared_ptr(void){

  key_t key;
  int shmid=-1;
```

```c
  key = ftok("/proc/cpuinfo",0x84);
  shmid = shmget(key, sizeof(shared_gpio_state), 0666 | IPC_CREAT);

  if (shmid != -1){
    DPRINTF("Created shared memory segment for rpi_gpio state\n");
  }
  else{
    DPRINTF("Failed to create shared memory segment. Error: %s\n",
strerror(errno));
  }

  return shmat(shmid, (void *)0, 0);

}

/* Iniitialize the device memory and sysbus connection
*/
static int rpi_gpio_initfn(SysBusDevice *sbd)
{
    DeviceState *dev = DEVICE(sbd);
    RPI_GPIO_State *s = RPI_GPIO(dev);

    memory_region_init_io(&s->iomem, OBJECT(s), &rpi_gpio_ops, s,
                          "rpi_gpio", 0x00B1);
    sysbus_init_mmio(sbd, &s->iomem);
    qdev_init_gpio_in(dev, rpi_gpio_set, 54);
    qdev_init_gpio_out(dev, s->out, 54);

    s->shm = get_shared_ptr();

    return 0;
}

static void rpi_gpio_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    SysBusDeviceClass *k = SYS_BUS_DEVICE_CLASS(klass);

    k->init = rpi_gpio_initfn;
    dc->vmsd = &vmstate_rpi_gpio;
    dc->reset = &rpi_gpio_reset;
}

static void rpi_gpio_init(Object *obj)
{
  DPRINTF("Initialized RPI_GPIO device\n");
}

static const TypeInfo rpi_gpio_info = {
    .name         = TYPE_RPI_GPIO,
    .parent       = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(RPI_GPIO_State),
    .instance_init = rpi_gpio_init,
    .class_init   = rpi_gpio_class_init,
};
```

```
static void rpi_gpio_register_types(void)
{
    type_register_static(&rpi_gpio_info);
}

type_init(rpi_gpio_register_types)
```

APPENDIX B: SOURCE CODE FOR QT DESIGNER WIDGETS

```
/* Also available at http://github.com/evplatt/mse_report/qt */
```

LED Widget

File LED.h

```cpp
#include <QtDesigner/QtDesigner>
#include <QWidget>

class QTimer;

class QDESIGNER_WIDGET_EXPORT LED : public QWidget
{
      Q_OBJECT

      Q_PROPERTY(double diameter READ diameter WRITE setDiameter)
      Q_PROPERTY(QColor color READ color WRITE setColor)
      Q_PROPERTY(Qt::Alignment alignment READ alignment WRITE setAlignment)
      Q_PROPERTY(bool state READ state WRITE setState)
      Q_PROPERTY(int gpio_pin READ gpio_pin WRITE set_gpio_pin)
      Q_PROPERTY(int refresh_rate READ refresh_rate WRITE set_refresh_rate)

public:
      explicit LED(QWidget* parent=0);
      ~LED();

      double diameter() const;
      void setDiameter(double diameter);

      QColor color() const;
      void setColor(const QColor& color);

      Qt::Alignment alignment() const;
      void setAlignment(Qt::Alignment alignment);

   bool state() const;

      void set_gpio_pin(int gpio_pin);
      void set_refresh_rate(int refresh_rate);
      int gpio_pin() const;
      int refresh_rate() const;
      void connect_gpio();
    int get_gpio_pin_function();

public slots:
      void setState(bool state);
      void gpio_refresh();

public:
      int heightForWidth(int width) const;
      QSize sizeHint() const;
      QSize minimumSizeHint() const;

protected:
```

```cpp
        void paintEvent(QPaintEvent* event);

private:
        double diameter_;
        QColor color_;
        Qt::Alignment alignment_;
        bool initialState_;
        bool state_;

        int gpio_pin_;
        int refresh_rate_;

        //
        // Pixels per mm for x and y...
        //
        int pixX_, pixY_;

        //
        // Scaled values for x and y diameter.
        //
        int diamX_, diamY_;

        QRadialGradient gradient_;
        QTimer* timer_;
        QTimer* gpio_timer_;

    struct shared_gpio_state {

      unsigned int GPFSEL0;
      unsigned int GPFSEL1;
      unsigned int GPFSEL2;
      unsigned int GPFSEL3;
      unsigned int GPFSEL4;
      unsigned int GPFSEL5;
      unsigned int GPLEV0;
      unsigned int GPLEV1;
      unsigned int outstate0;
      unsigned int outstate1;

    };

    const int gpio_to_bcm2835_map[8] = {17,18,21,22,23,24,25,4};

    shared_gpio_state *gpio_state;

};



File LED.cpp

#include <math.h>


#include <QPainter>
#include <QGradient>
```

```cpp
#include <QPaintDevice>
#include <QTimer>
#include <QDebug>
#include <sys/shm.h>
#include "LED.h"

LED::
LED(QWidget* parent) :
        QWidget(parent),
        diameter_(5),
        color_(QColor("red")),
        alignment_(Qt::AlignCenter),
        initialState_(true),
        state_(true),
        gpio_pin_(0),
        refresh_rate_(10)
{

    setDiameter(diameter_);
    set_gpio_pin(gpio_pin_);
    set_refresh_rate(refresh_rate_);

    connect_gpio();

    gpio_timer_ = new QTimer(this);
    connect(gpio_timer_, SIGNAL(timeout()), this, SLOT(gpio_refresh()));

    //start gpio readings
    gpio_timer_->start(refresh_rate_);

}

LED::
~LED()
{
}

void LED::
connect_gpio()
{

    key_t key = ftok("/proc/cpuinfo",0x84);
    int shmid = shmget(key, sizeof(shared_gpio_state), 0666);
    if (shmid == -1) qDebug() << "Error connecting to shared memory";
    gpio_state = (shared_gpio_state*)shmat(shmid, (void *)0, 0);

    update();

}

double LED::
diameter() const
{
        return diameter_;
}

void LED::
```

```cpp
setDiameter(double diameter)
{
    diameter_ = diameter;

    pixX_ = round(double(height())/heightMM());
    pixY_ = round(double(width())/widthMM());

    diamX_ = diameter_*pixX_;
    diamY_ = diameter_*pixY_;

    update();
}


QColor LED::
color() const
{
    return color_;
}

void LED::
setColor(const QColor& color)
{
    color_ = color;
    update();
}

Qt::Alignment LED::
alignment() const
{
    return alignment_;
}

void LED::
setAlignment(Qt::Alignment alignment)
{
    alignment_ = alignment;

    update();
}

void LED::
set_gpio_pin(int gpio_pin)
{
    gpio_pin_ = gpio_pin;

    update();
}

void LED::
set_refresh_rate(int refresh_rate)
{
    refresh_rate_ = refresh_rate;

    update();
}
```

```cpp
void LED::
setState(bool state)
{
      state_ = state;
      update();
}


void LED::
gpio_refresh()
{
    //make sure it's an output then set the LED state
    if (get_gpio_pin_function() == 1) state_ = ((gpio_state->outstate0 & (1
<< gpio_to_bcm2835_map[gpio_pin_])) > 0);
    else state_ = false;

    update();
}

int LED::
heightForWidth(int width) const
{
      return width;
}

QSize LED::
sizeHint() const
{
      return QSize(diamX_, diamY_);
}

QSize LED::
minimumSizeHint() const
{
      return QSize(diamX_, diamY_);
}

void LED::
paintEvent(QPaintEvent *event)
{
    Q_UNUSED(event);

    QPainter p(this);

    QRect geo = geometry();
    int width = geo.width();
    int height = geo.height();

    int x=0, y=0;
    if ( alignment_ & Qt::AlignLeft )
          x = 0;
    else if ( alignment_ & Qt::AlignRight )
          x = width-diamX_;
    else if ( alignment_ & Qt::AlignHCenter )
          x = (width-diamX_)/2;
    else if ( alignment_ & Qt::AlignJustify )
          x = 0;
```

```cpp
        if ( alignment_ & Qt::AlignTop )
                y = 0;
        else if ( alignment_ & Qt::AlignBottom )
                y = height-diamY_;
        else if ( alignment_ & Qt::AlignVCenter )
                y = (height-diamY_)/2;

        QRadialGradient g(x+diamX_/2, y+diamY_/2, diamX_*0.4,
                diamX_*0.4, diamY_*0.4);

        g.setColorAt(0, Qt::white);
        if ( state_ )
                g.setColorAt(1, color_);
        else
                g.setColorAt(1, Qt::black);
        QBrush brush(g);

        p.setPen(color_);
        p.setRenderHint(QPainter::Antialiasing, true);
        p.setBrush(brush);
        p.drawEllipse(x, y, diamX_-1, diamY_-1);

}

bool LED::
state() const
{
    return state_;
}

int LED::
gpio_pin() const
{
    return gpio_pin_;
}

int LED::
refresh_rate() const
{
    return refresh_rate_;
}

int LED::
get_gpio_pin_function(){

  int pin = gpio_to_bcm2835_map[gpio_pin_];

  if (pin<10)            return ((gpio_state->GPFSEL0 >> (3*pin))      &
0x7);
  if (pin>=10 && pin<20) return ((gpio_state->GPFSEL1 >> (3*(pin-10))) &
0x7);
  if (pin>=20 && pin<30) return ((gpio_state->GPFSEL2 >> (3*(pin-20))) &
0x7);
  if (pin>=30 && pin<40) return ((gpio_state->GPFSEL3 >> (3*(pin-30))) &
0x7);
```

```c
  if (pin>=40 && pin<50) return ((gpio_state->GPFSEL4 >> (3*(pin-40))) &
0x7);
  return ((gpio_state->GPFSEL5 >> (3*(pin-50))) & 0x7);

}
```

Button Widget

<u>File button.h</u>

```cpp
#include <QObject>
#include <QWidget>
#include <QPushButton>
#include <QMouseEvent>

class GPIOButton : public QPushButton
{
    Q_OBJECT

    Q_PROPERTY(int gpio_pin READ gpio_pin WRITE set_gpio_pin)
    Q_PROPERTY(int click_duration READ click_duration WRITE
set_click_duration)

public:
    GPIOButton(QWidget *parent = 0);
    ~GPIOButton();

    int gpio_pin();
    int click_duration();
    void set_gpio_pin(int pin);
    void set_click_duration(int duration);
    void set_gpio(bool state);
    int get_gpio_pin_function();

public slots:
  void click_timeout();

protected:
    void mousePressEvent(QMouseEvent *e);


private:

    int gpio_pin_;
    int click_duration_;

    struct shared_gpio_state {

      unsigned int GPFSEL0;
      unsigned int GPFSEL1;
      unsigned int GPFSEL2;
      unsigned int GPFSEL3;
      unsigned int GPFSEL4;
      unsigned int GPFSEL5;
      unsigned int GPLEV0;
      unsigned int GPLEV1;
      unsigned int outstate0;
      unsigned int outstate1;

    };
    shared_gpio_state *gpio_state;
```

```cpp
    const int gpio_to_bcm2835_map[8] = {17,18,21,22,23,24,25,4};


};



File button.cpp

#include <QApplication>
#include <string>
#include <QTimer>
#include <QDebug>
#include <sys/shm.h>
#include "GPIOButton.h"

GPIOButton::GPIOButton(QWidget *parent) : QPushButton(parent),
    gpio_pin_(0),
    click_duration_(100)
{

    //Connect to GPIO shared memory
    key_t key = ftok("/proc/cpuinfo",0x84);
    int shmid = shmget(key, sizeof(shared_gpio_state), 0666);
    gpio_state = (shared_gpio_state*)shmat(shmid, (void *)0, 0);

    set_gpio_pin(gpio_pin_);
    set_click_duration(click_duration_);

}


GPIOButton::~GPIOButton(){}


int GPIOButton::gpio_pin(){
    return gpio_pin_;
}

int GPIOButton::click_duration(){
    return click_duration_;
}

void GPIOButton::set_gpio_pin(int pin){
    gpio_pin_=pin;
}

void GPIOButton::set_click_duration(int duration){
    click_duration_=duration;
}


void GPIOButton::set_gpio(bool state){

   if (get_gpio_pin_function() == 0){
       qDebug() << "Set pin " << gpio_pin_ << " to " << (state?"1":"0");
        if (state)
```

40

```cpp
            gpio_state->GPLEV0 |= (1 << gpio_to_bcm2835_map[gpio_pin_]);
        else
            gpio_state->GPLEV0 &= (~(1 << gpio_to_bcm2835_map[gpio_pin_]));
    }

    update();
}


void GPIOButton::mousePressEvent(QMouseEvent *e) {
    if(e->button() == Qt::LeftButton) {
        set_gpio(true);
        QTimer::singleShot(click_duration_, this, SLOT(click_timeout()));
    }
}

void GPIOButton::click_timeout(){

    set_gpio(false);

}

int GPIOButton::
get_gpio_pin_function(){

  int pin = gpio_to_bcm2835_map[gpio_pin_];

  if (pin<10)            return ((gpio_state->GPFSEL0 >> (3*pin))      &
0x7);
  if (pin>=10 && pin<20) return ((gpio_state->GPFSEL1 >> (3*(pin-10))) &
0x7);
  if (pin>=20 && pin<30) return ((gpio_state->GPFSEL2 >> (3*(pin-20))) &
0x7);
  if (pin>=30 && pin<40) return ((gpio_state->GPFSEL3 >> (3*(pin-30))) &
0x7);
  if (pin>=40 && pin<50) return ((gpio_state->GPFSEL4 >> (3*(pin-40))) &
0x7);
  return ((gpio_state->GPFSEL5 >> (3*(pin-50))) & 0x7);

}
```

APPENDIX C: WIRINGPI LIBRARY MODIFICATION

The following was added to the piBoardRev() function which normally inspects /proc/cpuinfo to find information on what type of Raspberry Pi is in use and what the board revision is.

```c
/* Also available at http://github.com/evplatt/mse_report/wiringEmuPi */

/* Check for QEMU platform by checking the hard disk ID *
  d = opendir("/dev/disk/by-id");
  if (d){

    while ((dir = readdir(d)) != NULL){

      if (strstr (dir->d_name, "QEMU") != NULL){

        // Found a QEMU drive

        if (wiringPiDebug)
          printf ("piboardRev: Found a QEMU filesystem.\n") ;

        if ((homedir = getenv("HOME")) == NULL) {
          homedir = getpwuid(getuid())->pw_dir;
        }

        emucfg_path = malloc(120);
        strcpy(emucfg_path,homedir);
        strcat(emucfg_path, "/.emupi");

        if ((emucfg = fopen (emucfg_path, "r")) == NULL)
          piEmuCfgInfo();

        while (fgets (line, 120, emucfg) != NULL){

          if (strstr (line, "board_rev") != NULL) {

            if (wiringPiDebug)
              printf ("piboardRev: Found board_rev line in .wiringEmuPi:
                      %s\n",line);

            // Scan past the '='
            for (c = line ; *c ; ++c)
              if (*c == '=')
                break ;
            if (*c != '=')
             piBoardRevOops ("Bogus \"board_rev\" line in .wiringEmuPi (no
                      '=')");
            // Skip spaces
            ++c ;
            while (isspace (*c))
              ++c ;
            // Check board_rev
            if (*c != 0x31 && *c != 0x32)
```

42

```
                    piBoardRevOops ("Invalid board_rev in .wiringEmuPi (must be 1
                          or 2)");

               return boardRev = (*c-0x30);

          }

        }

      }
    }
    closedir(d);
  }
```

The following function provides a description of the config file that is required when running under a QEMU emulation.

```
static void piEmuCfgInfo(){

  fprintf (stderr, "Under QEMU, a configuration file is required in your home
                    folder with filename .emupi\n");
  fprintf (stderr, "The file must have the following options defined:\n");
  fprintf (stderr, "  -> board_rev = [board revision]\n");
  fprintf (stderr, "  -> board_id  = [board id]\n\n");
  fprintf (stderr, "board revision [1 or 2]:\n");
  fprintf (stderr, "  Original Pi ver 1:  board revision = 1\n");
  fprintf (stderr, "  Original Pi ver 2 or newer Pi: board revision =
                    2\n\n");
  fprintf (stderr, "board id [0002-0015]:\n\n");
  fprintf (stderr, "  0000 - Error\n");
  fprintf (stderr, "  0001 - Not used\n\n");
  fprintf (stderr, "  Original Pi boards:\n");
  fprintf (stderr, "    0002 - Model B,  Rev 1,   256MB, Egoman\n");
  fprintf (stderr, "    0003 - Model B,  Rev 1.1, 256MB, Egoman, Fuses/D14
                    removed.\n\n");
  fprintf (stderr, "  Newer Pi's with remapped GPIO:\n");
  fprintf (stderr, "    0004 - Model B,  Rev 2,   256MB, Sony\n");
  fprintf (stderr, "    0005 - Model B,  Rev 2,   256MB, Qisda\n");
  fprintf (stderr, "    0006 - Model B,  Rev 2,   256MB, Egoman\n");
  fprintf (stderr, "    0007 - Model A,  Rev 2,   256MB, Egoman\n");
  fprintf (stderr, "    0008 - Model A,  Rev 2,   256MB, Sony\n");
  fprintf (stderr, "    0009 - Model A,  Rev 2,   256MB, Qisda\n");
  fprintf (stderr, "    000d - Model B,  Rev 2,   512MB, Egoman (Red Pi, Blue
                    Pi?)\n");
  fprintf (stderr, "    000e - Model B,  Rev 2,   512MB, Sony\n");
  fprintf (stderr, "    000f - Model B,  Rev 2,   512MB, Qisda\n");
  fprintf (stderr, "    0010 - Model B+, Rev 1.2, 512MB, Sony\n");
  fprintf (stderr, "    0011 - Pi CM,    Rev 1.2, 512MB, Sony\n");
  fprintf (stderr, "    0012 - Model A+  Rev 1.2, 256MB, Sony\n");
  fprintf (stderr, "    0014 - Pi CM,    Rev 1.1, 512MB, Sony (Actual
                    Revision might be different)\n");
  fprintf (stderr, "    0015 - Model A+  Rev 1.1, 256MB, Sony\n");
```

```
    exit (EXIT_FAILURE) ;

}
```

APPENDIX D: SOURCE CODE FOR TEST GPIO PROGRAM

```c
/* Also available at http://github.com/evplatt/mse_report/testloop */

#include <stdlib.h>
#include <stdio.h>
#include "wiringPi.h" /* modified version of wiringPi library */

int main (void){

    wiringPiSetup();
    int i;

    for (i=0; i<4; i++) pinMode(i,OUTPUT);
    for (i=4; i<8; i++) pinMode(i,INPUT);

    while(1){
        for (i=4; i<8; i++){
            digitalWrite(i-4,digitalRead(i)); /* copy input level to
output */
        }
    }
}
```

Bibliography

1. Meyer, David. "This $35 Computer Just Passed a Major Sales Milestone." *Fortune.com*. TIME, 08 Sept. 2016. Web. 29 Oct. 2016. <http://fortune.com/2016/09/08/raspberry-pi-10-million/>.

2. "GPIO Programming Question." *Raspberry Pi Forums - General Discussion*. The Raspberry Pi Foundation, 7 Feb. 2012. Web. 30 Oct. 2016. <https://www.raspberrypi.org/forums/viewtopic.php?f=63&t=2946&sid=2d34adba88b7d 8179c3b11e71ed68c08>.

3. "QEMU." *Bifferboard*. N.p., n.d. Web. 17 Nov. 2016. <https://sites.google.com/site/bifferboard/Home/howto/qemu>.

4. "GPIO Source Code Listing." *QEMU Git Repository*. N.p., n.d. Web. 17 Nov. 2016. <http://git.qemu.org/?p=qemu.git;a=tree;f=hw/gpio;hb=b0bcc86d2a87456f5a276f941dc7 75b265b309cf>

5. Bellard, Fabrice. *QEMU Emulator User Documentation*. N.p., n.d. Web. 17 Nov. 2016. <http://wiki.qemu.org/download/qemu-doc.html>.

6. "System on a Chip." *Wikipedia*. Wikimedia Foundation, n.d. Web. 17 Nov. 2016. <https://en.wikipedia.org/wiki/System_on_a_chip>.

7. "Raspberry Pi." *Wikipedia*. Wikimedia Foundation, n.d. Web. 17 Nov. 2016. <https://en.wikipedia.org/wiki/Raspberry_Pi>.

8. "ChangeLog/2.6" *QEMU Wiki*. N.p., n.d. Web. 17 Nov. 2016. <http://wiki.qemu.org/ChangeLog/2.6>.

9. Vyas, Dhruv. "Raspberry Pi Notes : Emulating Jessie Image with 4.1.7 Kernel." *Dexter's Lab: Yet Another Tech Blog*. N.p., 22 Nov. 2015. Web. 1 Oct. 2016. <http://dhruvvyas.com/blog/?p=49>.

10. "Versatile Application Baseboard for ARM926EJ-S: User Guide." (2011): ARM. Web. 15 Oct. 2016. <http://infocenter.arm.com/help/topic/com.arm.doc.dui0225d/DUI0225D_versatile_appli cation_baseboard_arm926ej_s_ug.pdf>.

11. *Raspbian*. The Raspberry Pi Foundation, n.d. Web. 17 Nov. 2016. <https://www.raspbian.org/>.

12. "QEMU Memory Management." *GitHub*. N.p., n.d. Web. 05 Oct. 2016. <https://github.com/android/platform_external_qemu/blob/master/docs/QEMU-MEMORY-MANAGEMENT.TXT>.

13. Armbruster, Markus. "QEMU's New Device Model Qdev." KVM Forum 2010. Boston, MA. 9 Aug. 2010. Web. 5 Oct. 2016. <http://www.linux-kvm.org/images/f/fe/2010-forum-armbru-qdev.pdf>.

14. Bellard, Fabrice. "Features/QOM." *QEMU RSS*. N.d. Web. 29 Oct. 2016. <http://wiki.qemu.org/Features/QOM>.

15. "Product: The IDE." *Qt*. The Qt Company, n.d. Web. 15 Oct. 2016. <https://www.qt.io/ide/>.

16. "BCM2835 ARM Peripherals." *Broadcom SoC Documentation*. Broadcom Corporation, 06 Feb. 2012. Web. 01 Oct. 2016. <https://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>.

17. "GPIO: Raspberry Pi Models A and B." *Raspberry Pi Documentation*. Raspberry Pi Foundation, n.d. Web. 1 Oct. 2016. <https://www.raspberrypi.org/documentation/usage/gpio/README.md>.

18. Henderson, Gordon. "Wiring Pi Reference." *Wiring Pi.*, n.d. Web. 1 Oct. 2016. <http://wiringpi.com/reference/>.

19. Vugenfirer, Yan. "QEMU and Device Emulation." *Guest Drivers and USB Redirection Framework (USBDK).QEMU AND DEVICE EMULATION*: Daynix Computing LTD, 15 Dec. 2014. Web. 12 Oct. 2016. <http://logtel-conferences.com/Portals/2/09.%20QEMU.pdf>.

20. "Custom Widget Plugin Example." *Qt Designer Manual*. The Qt Company, n.d. Web. 1 Nov. 2016. <http://doc.qt.io/qt-5/qtdesigner-customwidgetplugin-example.html>.