XinuPi: Porting a Lightweight Educational Operating System to the Raspberry Pi

Eric Biggers
Department of Mathematics, Statistics, and Computer Science
Macalester College
1600 Grand Avenue
Saint Paul, Minnesota 55105-7045
ebiggers@macalester.edu

Farzeen Harunani, Tyler Much, Dennis Brylow
Department of Mathematics, Statistics, and Computer Science
Marquette University
1313 W. Wisconsin Avenue
Milwaukee, WI 53201-1881
{farzeen.harunani, tyler.much, dennis.brylow}@mu.edu

ABSTRACT

The Raspberry Pi is a credit-card sized computer, designed to support hands-on computer science education activities with minimal hardware cost. The Pi's low price, powerful ARM-based processor, and rich set of built-in peripherals has made it an attractive platform for the gamut of hobbyists, researchers, and educators. However, for those interested in embedded systems education, the Raspberry Pi presents a unique opportunity to build curricula introducing time-oriented, reactive, and embedded system software using a mass-produced embedded platform with broad appeal.

Despite the growing popularity of the Pi platform for computer science education, the device commonly runs a Linux-based software stack that is, in practice, opaque except to experienced developers. In addition, important hardware on the device remains poorly documented. This paper presents a port of the time-tested Embedded Xinu operating system to the Raspberry Pi, combining a commodity embedded processor with a lightweight kernel designed to support hands-on pedagogy at the lowest levels of the software system stack.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

Keywords

Raspberry Pi, Embedded Xinu, Embedded systems education

1. INTRODUCTION

The Raspberry Pi is a credit-card sized computer designed to be plugged into a keyboard and TV or HDMI monitor. It is intended to be used for educational purposes, such as teaching programming with Python, and it normally runs a Linux-based operating system, such as the Debian-based "Raspbian" or Arch Linux ARM [14]. There are currently two major versions of the Raspberry Pi: the Model A and Model B, which cost $25 and $35 respectively.

In aspects such as cost, convenience, and community, the Raspberry Pi hardware is well-suited for exploring low-level computer science concepts, such as operating systems, networking, and embedded systems. However, lack of documentation has limited software support for this platform to either the very simple (standalone assembly language) or the very complex (full-fledged Linux kernel), with very little available in the productive middle of the spectrum. Over 1 million Raspberry Pi boards were sold as by January 2013[13], but key aspects of the hardware, such as the USB controller (essential for network and keyboard support) and the mailbox subsystem (essential for programming the graphics co-processor), remain effectively undocumented except by sparsely commented code within vendor-contributed Linux device drivers. In cases where documentation is available, such as Broadcom's *BCM2835 ARM Peripherals* [5], some of the descriptions contains critical errors and omissions that, in our experience, have limited their usefulness.

XinuPi, our port of Embedded Xinu[9] to the Raspberry Pi, provides a cost-effective, usable, understandable platform for undergraduate computer science students to experiment with hardware/software interactions. XinuPi uses an elegant, well-documented kernel design with support for key hardware peripherals such as graphics, USB, and networking, yet is several orders of magnitude smaller than the most compact Linux-based software stacks that run on this device. XinuPi offers a unique point in the curriculum design space, allowing low-cost, multi-use laboratories equipped with Pi boards to support both the plethora of Python-, Scratch- and Linux-based educational activities already in use *and* low-level, embedded activities with access to GPIO pins, fine-grained clock facilities, a network stack, and the graphics processing unit. No hardware modifications are required to run XinuPi, so the same Raspberry Pi board can be easily retasked for higher-level activities with the swap of an SD card and a power cycle.

The source code of XinuPi, documentation, and class-tested laboratory activities from previous ports of Embedded Xinu are all freely available under a BSD-style license at `http://xinu.mscs.mu.edu`.

2. PRIOR AND RELATED WORK

The Raspberry Pi is now easy to obtain, and is used by a broad spectrum of hobbyists, researchers and educators. Applications of Raspberry Pis range from computer-controlled stage lighting[6], automatic coffee-brewing systems[16], and home entertainment systems[20]. Many computing educators have expressed interest in using the Raspberry Pi in their classrooms[4], whilst others have designed entire classes around the Pi.

Cambridge University has developed a series of hands-on Raspberry Pi exercises that builds a simple command line interface[11]. Their target is a single-thread assembly language executable which does not provide many of the core features of an operating system, such as concurrent threads, device drivers, or interrupts.

Raspberry Pi educational resources include programming environments such as Scratch and Python, which are fully implemented in the vendor-provided Raspian distribution, as well as a user guide written by co-creators of the Pi for beginners[15]. Similarly inexpensive embedded systems, such as the Arduino board, are commonly used as teaching resources beyond simply systems education. For example, high school teachers are using the Arduino to take creative approaches in teaching computational sciences. A recent example is the LilyPad e-textile approach, in which students sew textiles together to create programs out of quilts[17].

The Raspberry Pi also has a vibrant user community, as can be seen by the constant activity on the Raspberry Pi Forum. Many of the forum moderators are creators or designers of the board, who often share technical knowledge with the community. Because the Pi is so new, lessons have not been fully tested before classroom implementation; the forum represents a place to test lessons and applications.

Embedded Xinu has been used by a multitude of schools in a variety of classes to teach operating systems [9], networking [10], compilers [19], embedded systems [24], hardware systems [8], and software testing [21]. As a research platform, it has supported novel systems work ranging from IP telephony [22] to distributed, many-core operating systems [28].

While other groups have begun efforts to port Xinu to the Raspberry Pi[7], we believe our effort is the first successful project that ports all relevant functionality, including USB and networking.

3. PORTING XINU TO THE PI

The following subsections describe the hardware details relevant to porting a lightweight operating system to the Raspberry Pi. Much of this technical detail has been synthesized from unrelated sources or discovered by experimentation, and does not appear to be documented to this level in any prior work.

3.1 Hardware Overview

The core of the Raspberry Pi is the BCM2835 SoC (System on a Chip), which contains an ARM1176JZF-S CPU along with Broadcom's proprietary VideoCore co-processor and various ARM-accessible peripherals such as a system timer, interrupt controller, SD card interface, UART, and PCM audio interface [5]. USB 2.0 is supported through the "Synopsys DesignWare High-Speed USB 2.0 On-the-Go Controller". The Model A has one USB port, while the Model B has two USB ports. The Model B furthermore has an Ethernet port, but the actual hardware supporting it, likely for cost reasons, is in fact a USB device (namely, the "SMSC LAN9512 USB 2.0 Hub and 10/100 Ethernet Controller").

3.2 Booting and Startup

To boot the Raspberry Pi, one simply needs to apply power[1] after inserting a SD card containing appropriate boot files. The SD card must contain an MS-DOS-style disk label (partition table) with at least one FAT-formatted partition whose root directory contains three "binary blobs" provided by Broadcom (`start.elf`, `bootcode.bin`, and `loader.bin`) along with the actual kernel that gets started on the ARM processor (`kernel.img`). The reason for this somewhat inflexible boot method is that the VideoCore co-processor is in charge of the boot process. However, an optional file `config.txt` allows users to customize certain parameters, such as the memory split between the ARM and VideoCore.

After the VideoCore performs various tasks, `kernel.img` is loaded at physical memory address `0x8000` and the ARM begins execution at that address. The kernel is expected to be raw binary and not in a format such as ELF. To make Embedded Xinu comply with this boot protocol, we use a linker script that links the kernel to run at address `0x8000` and places the entry point at that address, then use `objcopy` to convert the kernel from ELF to raw binary.

The ARM startup code then must perform at least the following tasks:

1. Set up the ARM exception vector table, which is common to all ARM CPUs as described in [1], at physical memory address 0. The only handler strictly required for basic operating system functionality (including preemptive multitasking, but no memory protection etc.) is the IRQ handler, described in more detail later.

2. Enable desired CPU features, such as unaligned memory accesses, by reading and writing to the System Control Coprocessor (again, standard to the ARM architecture as described in [1]). The ARM contains L1 instruction and data caches that are initially disabled and can be enabled if desired.

3. Clear the `.bss` section of the kernel image for zero-initialized global variables.

4. Reserve memory for operating system use. In Embedded Xinu, we reserve space for the stack of the "nulluser" thread, which is the thread that runs the initial C startup code. We assign the rest of available physical memory to the "memheap", which is used for Embedded Xinu's dynamic memory allocator.

The bootloader also passes some additional information in the "atags" format[25] at address `0x100`. Embedded Xinu currently obtains the actual memory size and board serial number from there, but for basic operation no information is needed from these boot tags.

3.3 UART

Once code is running on the ARM following a successful boot, an essential task (at least for development) is to implement a way by which text can be "printed". Like many

[1]The Raspberry Pi is powered with 5V supplied through either the micro-USB port or the GPIO pins.

```
ctxsw:
        mrs r12, cpsr
        push {r0-r14}

        str sp, [r0]
        ldr sp, [r1]

        pop {r0-r12}
        msr cpsr_c, r12
        pop {lr, pc}
```

Figure 1: Context switch routine, written in ARM assembly language, for the Raspberry Pi port of Embedded Xinu. The corresponding C declaration is `void ctxsw(void **oldstackpp, void **newstackpp)`.

embedded devices, the Raspberry Pi has a Universal Asynchronous Receiver/Transmitter available that makes it possible to interact with the device through a serial connection. The connection can be made through a USB-to-serial converter with 3.3V logic levels. On the Model B, Tx is GPIO pin 8 and Rx is GPIO pin 10. No hardware modification is required.

In software, a PrimeCell (PL011)-compatible UART is available as a memory-mapped peripheral at physical address `0x20201000`. Software can operate this UART synchronously by polling it or asynchronously by configuring it to issue an interrupt when a character is received or when there is space to transmit a character. Rx and Tx FIFOs can be enabled but make asynchronous operation of the UART more complicated.

More details about the PL011 UART, including the register layout and interrupt handling, can be found in the Embedded Xinu driver[2], in ARM's documentation for PL011-compatible devices[2], or in Broadcom's documentation for the BCM2835 ARM peripherals[5].

3.4 Threads and Context Switching

Since Embedded Xinu supports multitasking, a critical next step is to implement support for threads and switching between them (context switching). On the Raspberry Pi, context switching deals only with the ARM CPU. Therefore the necessary bits and pieces are documented by ARM Ltd. and various other sources. For any CPU, context switching must save the registers of the currently executing thread and load the registers of the next thread. This includes general-purpose registers as well as the stack pointer and program counter. The ARM is no exception to this, but as shown in Figure 1, one must be careful to use the correct instructions and appropriately handle the CPSR (Current Program Status Register), which is not considered a general-purpose register.

In Embedded Xinu, creating new threads is implemented by constructing, from C code, a context record that can be switched to using the same code shown in Figure 1. The program counter ("pc") is set to the address of the new thread's starting routine. As per the standard ARM calling convention[3], up to the first four arguments are passed in registers r0 through r3, and any additional arguments are passed on the stack.

[2]See: `device/uart-pl011/`

IRQ line	Device
1	System timer output compare register 1
3	System timer output compare register 3
9	USB controller
55	PCM module
57	PL011 UART

Table 1: IRQ lines for devices described in this paper. Note: Embedded Xinu currently only uses lines 3, 9, and 57.

3.5 Interrupts

Threads by themselves are not very useful unless they can be interrupted by devices, such as the timer needed to implement preemptive multitasking. To receive interrupt requests (IRQs), software running on the Raspberry Pi must deal with two separate, yet interrelated mechanisms:

- The standard ARM architecture IRQ handling
- The BCM2835-specific interrupt controller

The ARM architecture mandates that the CPU can receive interrupts if and only if bit 7 of the Current Program Status Register is 0.[3] Therefore, software can write to this bit to enable and disable interrupts globally. When IRQs are enabled and an IRQ is received, the ARM automatically enters a special IRQ mode and jumps to the 7th word in the ARM exception vector table, which software must set up to contain a valid ARM instruction that jumps to the actual IRQ handling code[1]. To avoid disturbing the running thread, the IRQ handling code must save the values of any registers it may use, except those banked in IRQ mode. However, to simplify Embedded Xinu, we actually transition the processor out of IRQ mode immediately and run the IRQ handling code in the ARM CPU mode in which the kernel normally operates, which in Embedded Xinu is System mode.

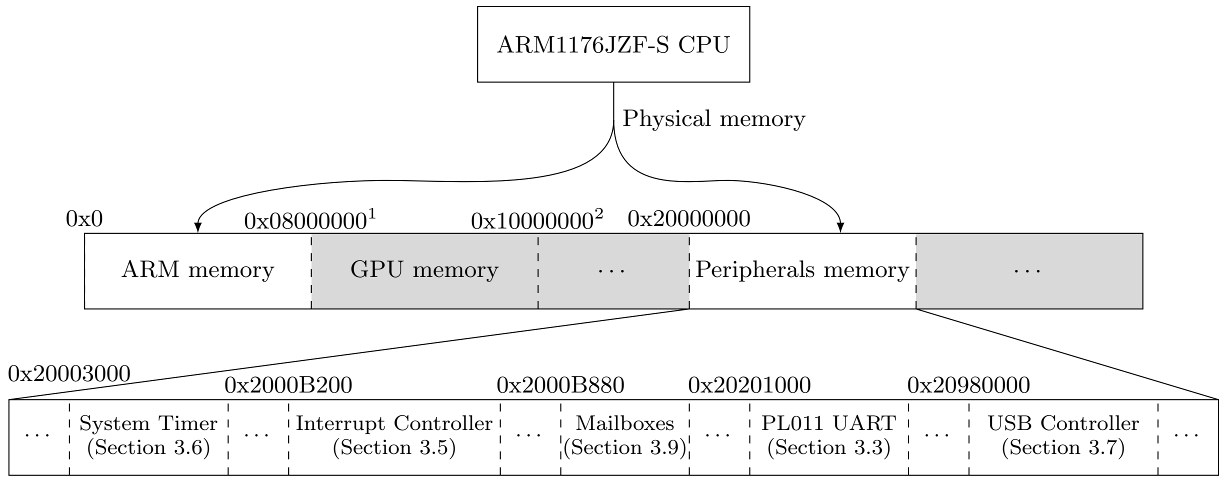
The global IRQ handling mechanism described above is standard to the ARM architecture. But to actually control interrupts from specific devices, software must use the BCM2835 SoC's interrupt controller, which is partially documented in [5]. The interface to this controller is a set of memory-mapped registers at physical address `0x2000B200` that allows software to enable or disable specific IRQ lines and check whether specific IRQs are pending. Since all IRQ lines are initially disabled, software *must* make use of the interrupt controller to receive any interrupts at all. Furthermore, software must know which interrupt line corresponds to which device, which is partially but not completely documented in [5]. For this reason, in Table 1 we list some of the important IRQ lines.

More details about the BCM2835 interrupt controller can be found in the Embedded Xinu source code.[4]

3.6 System Timer

[3]We do not consider the use of FIQs (Fast Interrupt Requests), which are an optional feature, not used by Embedded Xinu, for prioritizing interrupts from a specific device.

[4]See: `system/platforms/arm-rpi/dispatch.c`



[1] Actual split of memory between ARM and GPU is configurable and can be discovered at runtime.

[2] Actual total memory available to ARM and GPU depends on Raspberry Pi model and revision.

Figure 2: Logical view of the Raspberry Pi from software running on the ARM, primarily the physical memory map and locations of the various peripherals. As is the case on other ARM SoCs, most devices are directly accessible at fixed physical addresses. An exception is that USB devices can only be accessed indirectly, through the host controller. The Model A has no built-in USB devices, while the Model B includes a built-in SMSC LAN9512 USB 2.0 Hub and 10/100 Ethernet Controller (not visualized above).

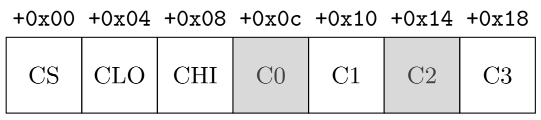


Figure 3: BCM2835 System Timer registers, based at physical memory address `0x20003000`. CLO and CHI make up the 64-bit free running counter. C0 through C3 are the output compare registers, but only C1 and C3 are available to the ARM.

Preemptive multitasking in Embedded Xinu requires a basic timer capable of interrupting the CPU after a programmable number of timer cycles have elapsed. Functions that must directly interface with the timer are the following:

- `void clkupdate(unsigned long cycles);` — schedules a timer interrupt to occur after the specified number of additional timer cycles have elapsed.
- `unsigned long clkcount(void);` — returns the current number of timer cycles.

The above does not include `clkhandler()`, the timer interrupt handler itself, which is platform-independent except for a call to `clkupdate()`.

On the Raspberry Pi, the BCM2835 System Timer provides the needed functionality. This timer, which is a memory-mapped peripheral, is partially documented in [5]. It has a 64-bit free-running counter, shown as CLO (Clock Low) and CHI (Clock High) in Figure 3, that contains a value that monotonically increases at a rate of 1,000,000 cycles per second. Directly following the 64-bit counter are four 32-bit "output compare" registers that software can set to trigger a timer interrupt to occur when CLO matches the programmed value. Directly preceding the 64-bit counter is the 32-bit Timer Control/Status register. In that register, software must set the bit corresponding to the output compare register index (0—3) to clear the corresponding interrupt.

A caveat not documented in [5] is that the VideoCore co-processor uses the output compare registers of indices 0 and 2 for itself, thereby leaving only the ones with indices 1 and 3 available for the ARM. However, Embedded Xinu only needs one such register anyway. Still, the presence of two separately-controlled microsecond-accuracy timers, using different interrupt lines, makes it possible to implement a variety of embedded and real-time software for the Raspberry Pi. Furthermore, [5] describes an additional timer, the "ARM Timer", that is also available to software.

3.7 Universal Serial Bus

As noted in Section 3.1, the Raspberry Pi contains support for Universal Serial Bus (USB) 2.0, and in fact USB support is *required* to access the Ethernet adapter on the Raspberry Pi Model B. Since Embedded Xinu did not contain support for USB, we implemented a relatively simple USB subsystem. It follows the standard design of partitioning functionality between the USB core, hub, and host controller drivers. The responsibilities of each driver are as follows:

- The USB core driver is platform-independent code responsible for maintaining the USB device model, performing device enumeration, and providing a framework in which USB device drivers can be written.
- The USB hub driver is a platform-independent USB device driver that controls hubs, which provide attach-

ment points for additional USB devices.

- The USB host controller driver is platform-dependent software responsible for interacting with the USB host controller hardware to actually send and receive data over the USB to or from a particular endpoint on a USB device.

Since USB is very complicated, not all relevant details, especially those pertaining to the USB core and hub drivers, can be mentioned here. Instead, we refer the reader to the USB 2.0 specification[12] and/or the Embedded Xinu source code[5]. Our focus here is instead on the *platform-dependent* portion of USB support, namely the USB host controller driver.

The USB specification does not standardize the interface USB host controllers present to software. Some host controllers present interfaces conforming to separate standards, such as OHCI or EHCI. However, the interface presented by the Synopsys DesignWare High-Speed USB 2.0 On-the-Go Controller used on the Raspberry Pi, which is a set of memory-mapped registers based at physical address `0x20980000`, does not conform to any particular standard, nor is there any publicly available documentation for it.

Therefore, the only way to support the Raspberry Pi's USB controller (other than by getting access to internal documentation, which may or may not even exist) was to glean the essential hardware details from other sources, primarily Synopsys Inc.'s Linux driver for the device[26]. This is, unfortunately, a difficult task because Synopsys' driver contains approximately 37,000 lines of code. From our analysis of the code, this extreme complexity is a result of a number of factors, including support for different modes of operation, different transfer types and speeds, multiple abstraction layers, support for multiple parameterizations of the same design in silicon, and support for power management features. There are many comments in the code, but no executive summary, that we could find, of what the code overall actually does.

Our contribution is a working, well-commented driver[6] to control the same hardware that runs in the lightweight Embedded Xinu environment and is approximately 20 times shorter than the corresponding Linux driver, mainly because our driver supports only a carefully-chosen subset of features and does away with unnecessary abstraction layers.

The entry point of our driver is `hcd_start()`, which must perform the following tasks to prepare the DesignWare Core to be ready to use:

1. Enable power to the Core by telling the VideoCore to do so, via the same memory-mapped "mailbox" message-passing mechanism the framebuffer driver (see Section 3.9) uses.

2. Reset the Core by setting a bit in the Core Reset Register, then waiting for the Core to clear it.

3. Enable DMA (Direct Memory Access) mode and configure dynamic FIFO locations and sizes. Explicitly configuring the dynamic FIFOs is required because the reset values are invalid. Failing to do so will result in silent memory corruption.

4. Software not performing USB transfers completely synchronously must enable interrupts from the Core, both in the AHB Configuration Register in the Core itself and in the BCM2835 interrupt controller described in Section 3.5. At minimum, software must enable Host Channel interrupts, which are required to be notified when USB transfers have completed, and Host Port interrupts, which are required to emulate the root hub.

After initialization, a minimal host controller driver need only support sending and receiving messages over the USB. Our driver implements this with the `hcd_submit_xfer_request()` function, which is passed a structure specifying a transfer to take place over the USB to or from a specific endpoint on a specific device. To actually perform such a transfer, software must, briefly, do the following:

1. Program one of the 8 available DesignWare Core *host channels* with the parameters of the USB transfer, including the transfer type, direction, size, and packet count, the device address and endpoint, and a pointer to a word-aligned buffer for DMA.

2. Wait for an interrupt to occur from the DesignWare Core (BCM2835 IRQ line 9).

3. Check the Core Interrupt Register, then the Host All Channels Interrupt Register, to determine which channel(s), if any, have halted.

4. Check the Channel Interrupt Register for each halted channel to determine whether the corresponding transfer completed successfully or failed.

The above is a simplified description and does not cover a multitude of special cases, such as the following:

- Transfers to or from low or full-speed devices, such as most USB HID devices (mice, keyboards, etc.) must be performed as a series of *split transactions*. To tell the DesignWare Core to execute such a transfer, special parameters must be set in the Split Control Register. Furthermore, on such transfers, the controller does not act autonomously to complete the full transfer. Instead, it halts the channel after every Start Split or Complete Split transaction, and software must take an appropriate action, such as retrying the transaction later.

- Although USB is a polled bus, as far as we can tell the DesignWare Core provides no support for hardware-based polling of devices. If an interrupt transfer is attempted from a device, such as a hub or keyboard, that has no data to send at the time, the DesignWare Core halts the channel and sets the "NAK response received" flag, thereby forcing software to explicitly delay for an appropriate interval for polling.

- The DesignWare Core seems to have some special scheduling requirements related to periodic transfers and frame boundaries which we do not yet fully understand. Our driver currently works around certain problems by automatically retrying certain transfers if they fail.

To provide more technical information about the undocumented Synopsys USB hardware, we have already provided

[5]See: `device/usb/`

[6]See: `system/platforms/arm-rpi/usb_dwc_hcd.c`

detailed documentation for the essential registers in Embedded Xinu's header file declaring them,[7] and we plan to further document the driver and hardware on the Embedded Xinu Wiki.[8] However, we can only document the functionality actually used by our driver, so we do not expect that our documentation by itself could be used to, for example, write a drop-in replacement for Synopsys' Linux driver that will not have any feature regressions.

3.8 Ethernet Support

The Raspberry Pi Model B's Ethernet port is part of the integrated SMSC LAN9512 USB 2.0 Hub and 10/100 Ethernet Controller. The hub component of the device is directly attached to the host port of the Synopsys USB controller. This hub contains three ports. Two are the ports physically available to plug devices into, while the third is a vendor-specific class device that implements the actual Ethernet Adapter. The software interface of the Ethernet Adapter device is not documented by SMSC, so here we provide a summary of it. However, do note that writing a driver for this device relies on USB support, which when not available is a much more difficult component to implement, especially when dealing with the undocumented USB hardware described in Section 3.7.

To configure the SMSC LAN9512 Ethernet device, software must read and write its control registers, which are accessible by sending vendor-specific class requests to its default control endpoint. To read a register, software must send a USB control message with `bRequest` set to `0xA1`, `bmRequestType` set to `0xc0`, `wValue` set to 0, `wIndex` set to the register offset, `wLength` set to 4, and the data buffer set to a 4-byte location into which to read the register's contents. Writing a register proceeds similarly, but `bRequest` must be `0xA0`, `bmRequestType` must be `0x40`, and the data buffer must specify the 4-byte value to write. At minimum, software must explicitly set a MAC address in the MAC address low and MAC address high registers (offsets `0x108` and `0x104`, respectively), then enable Tx and Rx by setting bits 2 and 3 in the MAC Control Register (offset `0x100`) and bit 2 in the Tx Configuration Register (offset `0x10`). We have documented these registers and more in Embedded Xinu's header file declaring them, and we plan to write additional documentation on the Embedded Xinu Wiki.

Finally, to actually send and receive packets, the SMSC LAN9512 Ethernet device has two USB bulk endpoints. One is oriented host-to-device and is used to send Ethernet packets; the other is oriented device-to-host and is used to receive Ethernet packets. Incoming Ethernet packets are prefixed by a 4-byte status field. Outgoing Ethernet packets must be prefixed by two 4-byte command fields. See the code for details.[9] For simplicity, we do not attempt to support advanced features, such as TCP/IP checksum offloading.

3.9 Framebuffer

In addition to USB and networking, another important Raspberry Pi hardware feature to support is graphics output via a framebuffer. Before pixels can be rendered, a communication channel between the ARM and the VideoCore (VC) must be initialized. On the Raspberry Pi, a "mailbox system" exists to simplify communication. Mailboxes are used on the board for communication between various systems, such as the power management system, the framebuffer, and the LEDs. In order to initialize the framebuffer device (the structure used by the VC to keep track of pixel data), one needs to create a framebuffer structure, write its address to the appropriate mailbox channel, and read from the status register. Much of the driver was influenced by the video driver provided by Broadcom.

The framebuffer structure[23] expected by the VC is a list of unsigned little endian 32 bit integers, corresponding to, in order:

- requested width of physical display
- requested height of physical display
- requested width of virtual display
- requested height of virtual display
- requested depth (in bits per pixel)
- pitch (bytes between rows; zero upon request)
- requested x offset of virtual framebuffer
- requested y offset of virtual framebuffer
- framebuffer address (zero upon request; failure if zero upon response)
- framebuffer size (zero upon request)

A more detailed outline of the steps required to initialize the framebuffer:

1. Create framebuffer structure, as above.
2. Read from status register until full flag is not set.
3. Write data (shifted into upper 28 bits) combined with channel (in lower four bits) to write register.
4. Read from status register until empty flag is not set.
5. Read data from read register. Ensure that lower four bits are desired channel number.
6. Upper 28 bits should be all zeroes, unless there was a read error.
7. Ensure that the VC has written both the framebuffer address and the framebuffer size into the structure. If these values remain zero, the VC has rejected the request.

It should be noted that the framebuffer communication occurs on channel 1 of mailbox 0. The mailbox 0 read register is located at `0x2000B880`, the write register is located at `0x2000B8A0`, and the status register is located at `0x2000B898`.

Once the framebuffer is correctly initialized, pixels can be rendered. Each pixel is a 32-bit memory-mapped value (will vary depending on requested depth). All pixels start off as zeroed values: zero transparency, zero red, zero green, zero blue. In order to change a pixel onscreen, a new 32-bit color value must be written to the pixel's location in memory, which can be calculated by adding the address of the framebuffer to an offset of:

$$(y * pitch) + (x * (depth/8))$$

In order to display characters onscreen, we created an array of the first 128 ASCII characters to use as an 8 x 12 monospace font. Each character is represented by 12 bytes, and each byte corresponds to a line of pixels. A bit value of zero is rendered in background color, and a bit value of one is rendered in foreground color. For example, to draw

[7]See: `system/platforms/arm-rpi/usb_dwc_regs.h`
[8]Available at `http://xinu.mscs.mu.edu`
[9]See: `device/smsc9512/`

a space character, no pixels need to be changed from background to foreground. Thus, a space can be represented as 12 bytes of zeroes.

In order to draw shapes quickly and with enough accuracy, we employed Bresenham's Algorithm for drawing lines, and a modified version of his algorithm, the Midpoint Algorithm, for drawing circles[27].

3.10 PCM Audio

In addition to graphics output described in Section 3.9, the Raspberry Pi supports sound output via HDMI, GPIO pins, or the 3.5mm TRS connector. Here we describe how a PCM sound signal can be sent over the GPIO pins.

The PCM module on the BCM2835 provides three output signals:

- Bit Clock (PCM_CLK)
- Frame Sync (PCM_FS)
- Serial Data Output (PCM_DOUT)

Each of these three signals is easily accessible from the GPIO pins. In order to enable output to the GPIO, however, the pins must be set to the appropriate alternate function. The PCM_CLK signal can be read on GPIO 18 on the P1 header when the pin is set to alternate function 0. The PCM_FS and PCM_DOUT signals can be read on GPIO 29 and GPIO 31, respectively, on the P5 header when the pins are set to alternate function 2.

The BCM2835 ARM Peripherals document[5] provides information about the registers and the overall PCM interface that is sufficient to initialize and configure the PCM module. The documentation specifies how to configure the module to accept an input clock but does not specify where the audio clock control registers are located in memory, nor does it provide any information about how to manipulate these registers. This is necessary information because without choosing, configuring, and enabling the input clock, the module will receive no driver clock signal and no output signals will be transmitted.

The registers used to configure the input clock are documented in a separate datasheet[18]. They include a clock control register and a clock divisor register. It is important when modifying these memory mapped registers that `0x5a` is written to the most significant 8 bits, otherwise no change will occur. Using the following settings:

- Clock source set to internal oscillator (19.2MHz)
- MASH set to 1
- Integer Divisor set to 13
- Fractional Divisor set to 2479
- Frame length of 32 (PCM_MODE Register)
- Frame Sync length of 16 (PCM_MODE Register)

one can measure the PCM_CLK and PCM_FS output signals running at 1.44MHz and 44.1kHz, respectively. These are the frequencies appropriate for producing standard 44.1kHz stereo audio.

4. FUTURE WORK

4.1 Laboratory Environment

As described in [9], Embedded Xinu has been deployed on the WRT54G family of embedded wireless routers to create a laboratory environment for teaching embedded operating systems. The overall design is that students can work on their custom Embedded Xinu kernels remotely for various assignments, then submit them to a central server to be automatically run on real backend hardware selected from an available pool. The setup is such that students can interactively use the backend's console remotely. Since the educational value of this approach has been demonstrated, one of the goals of porting Embedded Xinu to the Raspberry Pi is to use it for this purpose as a substitute for the WRT54G-family routers. This will have several advantages, including eliminating the need for hardware modifications, reducing per-unit cost, updating the platform to one that is in commercial production as of 2013, and using a more flexible platform that can easily be removed from the backend pool and used for other purposes simply by swapping out the SD card.

To configure a Raspberry Pi as a backend device, it must run a bootloader that can download a new kernel over the network and execute it. Since the Raspberry Pi firmware does not offer this functionality, our solution is to use a customized image of Embedded Xinu itself, booted from the SD card. As of this writing, all necessary functionality to make Embedded Xinu act as a network bootloader has been implemented. This includes Ethernet support, as described in Section 3.8, simple DHCP and TFTP clients, and special code to abandon the running kernel and transfer control to a new kernel. In the coming months we plan to actually set up such a backend pool, then document the setup on the Embedded Xinu Wiki and demonstrate its educational use in a university-level operating systems class.

4.2 Documentation

As Embedded Xinu is an educational operating system, it is critical that it be well-documented. To this end, the Embedded Xinu Wiki already documents the supported MIPS platforms. This complements the detailed API documentation generated from the source code. We plan to expand this documentation to include the Raspberry Pi port and improve the API documentation to better document all major Embedded Xinu components and be easier to navigate.

4.3 Keyboard-and-Monitor Setup

The major advantage of using the Raspberry Pi over the Linksys routers is increased flexibility. USB support and a framebuffer driver allow Xinu to act as an experimental embedded laboratory platform that does not rely on remote booting or serial output.

We have developed a simple LOGO-like Turtle Graphics environment for the board that can be used in K-12 classrooms to enhance lessons in areas such as angles and shapes, in algorithmic, sequential reasoning, or in RGB color mixing.

We have also achieved communication from the Dell L100 USB keyboard. Our goal is to complete the HID (Human Interface Device) driver and create a HID device structure that is simple enough for Xinu and allows for additional HID class device drivers to be built on top of it. We aim to implement a keyboard-specific device structure so that a USB keyboard can be read from like other devices in Xinu.

With this "standalone" setup, Embedded Xinu can escape the remote access and power management infrastructure currently in place for the Linksys routers and be better suited for smaller educational environments.

5. SUMMARY & CONCLUSIONS

We present XinuPi: Embedded Xinu running on the Raspberry Pi. This project takes advantage of new, inexpensive hardware explicitly designed for computing education, and simplifies the system programmer's view of the hardware to a level well-suited for undergraduate education.

This work builds upon an established body of Embedded Xinu curriculum materials that weaves hands-on embedded systems laboratories into core computer science courses ranging from introductory hardware systems through operating systems, networking, compiler construction, and embedded systems.

Finally, in the course of porting Embedded Xinu to the Pi board, we have learned a great deal about the platform that is not well-documented in any of the existing materials. We present those insights here as a reference for like-minded developers with an eye on developing their own software to run directly on the Raspberry Pi, without needing to rely on Linux to communicate to the hardware.

6. ACKNOWLEDGEMENTS

We would like to thank Jason Laqua, for his contributions to the XinuPi project, and Matthew Bajzek, for his support.

This work is supported in part by NSF grant CNS-REU-1063041.

7. REFERENCES

[1] ARM Ltd. *ARM Architecture Reference Manual*, 2005.

[2] ARM Ltd. *PrimeCell UART (PL011) Technical Reference Manual*, r1p4 edition, 2005.

[3] ARM Ltd. *Procedure Call Standard for the ARM Architecture*, November 2012.

[4] J. Black, J. Brodie, P. Curzon, C. Myketiak, P. W. McOwan, and L. R. Meagher. Making computing interesting to school students: teachers' perspectives. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, ITiCSE '13, pages 255–260, New York, NY, USA, 2013. ACM.

[5] Broadcom. *BCM2835 ARM Peripherals*, 2012.

[6] J. Brogdon. Control the Limelight with a Raspberry Pi. *Linux Journal*, 2013(229):84–94, 2013.

[7] J. S. Brown. Operating systems II syllabus. http://www.cs.rit.edu/ jsb/20123/OS2/syllabus.php, 2012.

[8] D. Brylow. An experimental laboratory environment for teaching embedded hardware systems. In *Proceedings of the 2007 workshop on Computer architecture education*, WCAE '07, pages 44–51, New York, NY, USA, 2007. ACM.

[9] D. Brylow. An experimental laboratory environment for teaching embedded operating systems. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 192–196, New York, NY, USA, 2008. ACM.

[10] D. Brylow and K. Thurow. Hands-on networking labs with embedded routers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 399–404, New York, NY, USA, 2011. ACM.

[11] A. Chadwick. Baking Pi. http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/, 2013.

[12] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Universal Serial Bus Specification, April 2000.

[13] Raspberry Pi Foundation. 500,000 Pis in Wales. http://www.raspberrypi.org/archives/3686, April 2013.

[14] Raspberry Pi Foundation. FAQs. http://www.raspberrypi.org/faqs, July 2013.

[15] G. Halfacree and E. Upton. *Raspberry Pi User Guide*. John Wiley and Sons, Chicester, England, 2012.

[16] M. Johnsen. MoccaPi. http://moccapi.blogspot.com/, 2013.

[17] Y. B. Kafai, K. Searle, E. Kaplan, D. Fields, E. Lee, and D. Lui. Cupcake cushions, scooby doo shirts, and soft boomboxes: e-textiles in high school to promote computational concepts, practices, and perceptions. In *Proceeding of the 44th ACM technical symposium on Computer science education*, SIGCSE '13, pages 311–316, New York, NY, USA, 2013. ACM.

[18] G. J. V. Loo. *BCM2835 Audio & PWM clocks*, February 2013.

[19] A. B. Mallen and D. Brylow. Compiler construction with a dash of concurrency and an embedded twist. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 161–168, New York, NY, USA, 2010. ACM.

[20] S. Nazarko. Raspbmc. http://www.raspbmc.com/, 2013.

[21] M. H. Netkow and D. Brylow. Xest: an automated framework for regression testing of embedded software. In *Proceedings of the 2010 Workshop on Embedded Systems Education*, WESE '10, pages 7:1–7:8, New York, NY, USA, 2010. ACM.

[22] K. Persohn and D. Brylow. Interactive real-time embedded systems education infused with applied internet telephony. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 199–204, 2011.

[23] N. J. Phillips. Raspberry Pi Firmware Wiki. https://github.com/raspberrypi/firmware/wiki, 2012.

[24] P. Ruth and D. Brylow. An experimental nexos laboratory using virtual xinu. In *Proceedings of the 2011 Frontiers in Education Conference*, FIE '11, pages S2E–1–1–S2E–6, Washington, DC, USA, 2011. IEEE Computer Society.

[25] V. Sanders. Booting ARM Linux. http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html, 2004.

[26] Synopsys, Inc. and additional contributors. Synopsys HS OTG Linux Software Driver. https://github.com/raspberrypi/linux/tree/rpi-3.6.y/drivers/usb/host/dwc_otg, 2013.

[27] A. Zingl. The beauty of bresenham's algorithm. http://members.chello.at/~easyfilter/bresenham.html, 2012.

[28] M. Ziwisky, K. Persohn, and D. Brylow. A down-to-earth educational operating system for up-in-the-cloud many-core architectures. *Trans. Comput. Educ.*, 13(1):4:1–4:12, Feb. 2013.