CORE
Metadata, citation and similar papers at core.ac.uk
Provided by DigitalCommons@Macalester College

Macalester College
DigitalCommons@Macalester College

Mathematics, Statistics, and Computer Science Honors Projects

Mathematics, Statistics, and Computer Science

5-2014

Porting the Embedded Xinu Operating System to the Raspberry Pi

Eric Biggers
Macalester College, ebiggers3@gmail.com

Follow this and additional works at: http://digitalcommons.macalester.edu/mathcs_honors


Part of the OS and Networks Commons, Software Engineering Commons, and the Systems Architecture Commons

Recommended Citation
Biggers, Eric, "Porting the Embedded Xinu Operating System to the Raspberry Pi" (2014). *Mathematics, Statistics, and Computer Science Honors Projects*. Paper 32.
http://digitalcommons.macalester.edu/mathcs_honors/32

This Honors Project is brought to you for free and open access by the Mathematics, Statistics, and Computer Science at DigitalCommons@Macalester College. It has been accepted for inclusion in Mathematics, Statistics, and Computer Science Honors Projects by an authorized administrator of DigitalCommons@Macalester College. For more information, please contact scholarpub@macalester.edu.

MACALESTER COLLEGE

HONORS PAPER IN COMPUTER SCIENCE

Porting the Embedded Xinu Operating System to the Raspberry Pi

---

Author:
Eric Biggers

Advisor:
Shilad Sen

May 5, 2014

Abstract

This thesis presents a port of a lightweight instructional operating system called Embedded Xinu to the Raspberry Pi. The Raspberry Pi, an inexpensive credit-card-sized computer, has attracted a large community of hobbyists, researchers, and educators since its release in 2012. However, the system-level software running on the Raspberry Pi has been restricted to two ends of a spectrum: complex modern operating systems such as Linux at one end, and very simple hobbyist operating systems or simple "bare-metal" programs at the other end. This project bridges this gap by porting the Embedded Xinu operating system to the Raspberry Pi. Although simple and designed for educational use, Embedded Xinu supports major features of modern operating systems such as preemptive multitasking and networking. This thesis also presents the addition of new optional features, such as USB support, to Embedded Xinu, and demonstrates major challenges that may arise when writing device drivers for modern hardware.

Contents

Chapter 1

Introduction

Operating systems are a fundamental part of modern computing. In the broadest sense, an operating system is an abstract machine on which application programs run, providing benefits such as code reuse, isolation from the underlying hardware, multitasking, and efficient resource sharing. The understanding of operating systems is a critical part of a computer science education [17]. But beneath the abstractions they provide, modern operating systems are highly complex software systems that frequently contain millions of lines of code, as is necessary to keep up with new hardware and features while still maintaining backwards compatibility with legacy hardware and applications. Although open source operating systems such as Linux are readily available for educational use, they are generally considered too complex and production-focused to be used as the basis for a broad overview of operating system functionality in an educational setting [9].

Instructional, or educational, operating systems provide a way for students to learn about operating systems and related topics in a hands-on manner without immediately plunging deep into highly complex operating systems such as Linux. In addition, an operating systems course incorporating an instructional operating system provides an interesting and practical alternative to a purely theoretical, book-based course without any hands-on activities.

Embedded Xinu is a contemporary instructional operating system based on the original Xinu operating system designed by Dr. Douglas Comer in 1984 [17] [22]. Developed primarily by students under the guidance of Dr. Dennis Brylow at Marquette University, Embedded Xinu is a flexible operating system that runs on both real and virtual hardware and has been adapted for use in variety of computer science courses, including operating systems [18], hardware systems [17], networking [19], and compiler construction [38]. Embedded Xinu is distinguished from the original Xinu by its target of modern embedded platforms, such as commercially available MIPS-based routers, as well as modernization of the codebase. As part of the NexOS Project, Marquette University, the University of Buffalo, and the University of Mississippi have established experimental laboratories for hands-on projects in embedded systems courses [45]. These laboratories rely on Embedded Xinu running on Linksys WRT54GL routers.

Although MIPS-based routers such as the Linksys WRT54GL are fitting hardware platforms for Embedded Xinu due to their low cost, embedded characteristics, and distinctive features such as multiple networking interfaces, they suffer from several limitations. Such routers are typically designed in such a way that low-level hardware access is not readily available to the end user. For example, before Embedded Xinu can run on the WRT54GL router, the user must perform a hardware modification to enable access to the integrated serial ports [17]. Indeed, buying, modifying, and setting up one or two dozen routers requires significant time, effort, and knowledge that not all instructors who might benefit from Embedded Xinu have.

My work brings Embedded Xinu to a new hardware platform, the Raspberry Pi, in a project called "XinuPi". The Raspberry Pi is a credit-card-sized computer designed by the Raspberry Pi Foundation, a nonprofit organization, to support hands-on computer science education with minimal hardware cost. While being priced at only $25 or $35 depending on the model, the Raspberry Pi offers modern hardware such as a fairly powerful CPU and GPU, memory card support, and USB support. It also offers GPIO (General Purpose Input/Output) pins for easy hardware interfacing and serial console access [40].

The combination of a low price, organizational backing, a rich set of hardware peripherals, and Linux support have attracted a large community of hobbyists, researchers, and educators to the Raspberry Pi since its release in 2012. Yet despite much interest in lower-level programming on the Raspberry Pi [4], much of the hardware has remained fairly inaccessible to programmers working outside a Linux environment. Consequently, "bare metal" projects on the Raspberry Pi have typically remained quite simple, and the platform has not yet been made available for operating systems education in the style of Xinu. My work helps alleviate this problem by providing and documenting a port of Embedded Xinu to the Raspberry Pi, which demonstrates a simple operating system several orders of magnitude smaller (in terms of code) than Linux running on the very same hardware.

XinuPi is also a practical demonstration of issues that arise when writing or porting an operating system to a new hardware platform, in particular one based on a modern system-on-a-chip designed for embedded systems.

My work on XinuPi began in June 2013 as part of the *Computation Across the Disciplines* REU (Research Experience for Undergraduates) program at Marquette University, in collaboration with Dr. Dennis Brylow and two of his students — Farzeen Harunani and Tyler Much. Throughout the summer of 2013 and with varying degrees of collaboration with the other contributors, I implemented or ported various operating system features including thread creation, context switching, interrupt handling, a USB subsystem, a TFTP client, a DHCP client, and a driver for SMSC LAN9512 USB Ethernet devices.

Towards the end of the summer, Farzeen, Tyler, Dr. Brylow, and I wrote an 8-page paper entitled *XinuPi: Porting a Lightweight Educational Operating System to the Raspberry Pi*. In October 2013, we presented our paper at the 2013 Workshop

on Embedded and Cyberphysical Systems Education (WESE), organized as part of Embedded Systems Week (ESWEEK) in Montreal, Canada [14]. Our paper briefly described our preliminary work with XinuPi, but in this thesis I go into much more detail and also cover later work.

During the fall of 2013, I made other improvements to Embedded Xinu, including documentation improvements, bug-fixes, and the implementation of a driver for USB keyboards. During this time I discussed the operating system with other developers on the private Embedded Xinu mailing list. However, no other developers made source code contributions during this time.

In March 2014, I completed a basic port of Embedded Xinu to the *arm-qemu* virtual platform. Although I do not describe the *arm-qemu* port in detail in this thesis, it borrowed heavily from the Raspberry Pi port ("XinuPi"), since both target ARM-based platforms. This shows that the Raspberry Pi port of Embedded Xinu has, in addition to accomplishing other goals, made it easier to support other platforms using the ARM processor architecture, which is commonly used in modern embedded systems.

At this stage in the project, XinuPi can now be used in various computer science courses. Some of the established Embedded Xinu curriculum materials can be reused, whereas others will need to be modified to take into account the Raspberry Pi hardware. Originally, Tyler had planned to work with Dr. Brylow to set up a laboratory of Raspberry Pis in the Systems Laboratory at Marquette University for use in their operating systems course in the spring of 2014. However, this seems not to have happened as of March 2014. Still, at least one other school has already made use of XinuPi. Section 9.3 contains more details on potential and actual educational use of XinuPi.

This thesis will begin with a more in-depth discussion of Embedded Xinu, without special focus on the Raspberry Pi. This will be followed by an in-depth discussion of the relevant Raspberry Pi hardware, then various chapters where I present my work to port or implement specific Embedded Xinu operating system components and features, primarily for the Raspberry Pi platform. This thesis will conclude with a discussion of other work — both on Embedded Xinu and external related work — and future steps.

Chapter 2

Embedded Xinu

This chapter presents background information on the Embedded Xinu operating system, most of which predates my work. However, I will mention my work when essential to understanding the current status of the Embedded Xinu project overall (and not just the Raspberry Pi support).

2.1 Design and history

2.1.1 Overview

The original Xinu (or XINU) operating system is an instructional operating system created by Dr. Douglas Comer at Purdue University in 1984 [17] [22]. The original code was written for the LSI-11 computer. "Xinu" is both "UNIX" spelled backwards and a recursive acronym standing for "Xinu is not UNIX". Although Xinu shares some characteristics with UNIX, it has its own unique design that emphasizes simplicity. Over the years, Xinu has been ported to new platforms, used at various colleges and universities, and even used in commercial products [23].

Embedded Xinu is a re-implementation of Xinu started by Dr. Dennis Brylow at Marquette University in 2006 [5]. The original goal of the Embedded Xinu project was to port Xinu to the MIPS architecture by specifically targeting Linksys WRT54GL routers. A secondary goal was to modernize the codebase through the use of ANSI C syntax rather than K&R C syntax, the use of de-facto standard modern tools such as the GCC compiler to build the operating system, and the use of modern documentation tools such as Doxygen. Following the original work with the WRT54GL, Embedded Xinu has also been ported to several other MIPS-based routers, QEMU [13] (using MIPSel or ARM emulation), a JavaScript x86 PC emulator, and the Raspberry Pi (this thesis).

Despite sharing some fundamental design elements, Embedded Xinu is in many ways very different from commonly used operating systems such as Linux and Windows. As an instructional operating system designed primarily for embedded hardware, it is

not intended for everyday use; rather, it is intended that users extend or modify the code for specific applications. Furthermore, as part of its simple design and embedded focus, there is no distinction between user-space and kernel-space; that is, Embedded Xinu applications are simply made part of the operating system itself, sometimes as commands in the interactive shell. Essentially, a build of Embedded Xinu is a single binary image, similar to the "kernel" of other operating systems. Users typically interact with Embedded Xinu through a command-line interface over a serial port, but a TELNET server is also available on platforms supporting networking. In addition, partly as a result of my work, a keyboard-and-monitor setup is now supported on the Raspberry Pi. There is currently no official support for filesystems.

Like most modern operating systems, Embedded Xinu is primarily written in the C programming language. Some very low-level, platform-specific subroutines are, by necessity, written in assembly language. Work has been done to port a Lua interpreter to Embedded Xinu [48], but currently C is still the primary development language due to its widespread use and its suitability to systems-level programming.

Embedded Xinu has been used in operating systems courses, where students receive a stripped-down version of the code and work to implement, by themselves, important operating system features [18]. Embedded Xinu has also been used in a hardware systems course because it makes available low-level hardware access and provides an engaging environment for assembly-language programming for a modern RISC architecture [17].[1] Mallen and Brylow [38] describe a compiler construction course that ties the construction of a compiler for the "MiniJava" programming language to an Embedded Xinu back-end supporting concurrent threads. Brylow and Thurow [19] describe how the networking features and router support in Embedded Xinu have made possible a course in computer networking that features hands-on activities such as building a custom network stack.

As part of its instructional focus, Embedded Xinu focuses on easy-to-understand, well-documented code and modularity. The goal of the project is not to simply add more features, but rather to maintain only the most useful features at a given time and maintain a clean, well-documented design that can be understood in a reasonable amount of time. The core of the operating system — the code built into the kernel regardless of the platform — is quite small, since most functionality is in modules[2] that can be disabled at compilation time. As such, the operating system can still be easily stripped down to its core, despite the addition of certain new features, such as USB, as described in this thesis.

[1] RISC (Reduced Instruction Set Computing) microprocessor architectures, such as MIPS and ARM, are characterized by having simpler instruction sets than CISC (Complex Instruction Set Computing) architectures, such as x86.

[2] The use of the word "modules" here is not meant to imply that Embedded Xinu supports dynamically loadable kernel modules, like Linux does.

2.1.2 Threads and multitasking

A running instance of Embedded Xinu consists of one or more *threads*, each of which is an independent flow of control within the operating system. Unlike more complex operating systems, no distinction is made between kernel and user threads. Furthermore, there is no distinction between "processes" and "threads" — there are only threads, and all share the same memory space.

Although only one CPU is supported[3], Embedded Xinu still allows multiple threads to execute concurrently through the use of *preemptive multitasking*, which is a commonly used technique that provides the illusion that multiple threads are executing at the same time on the same processor. Preemptive multitasking works by configuring a hardware timer to periodically interrupt the CPU. When such an interrupt occurs, the operating system has the opportunity to reschedule the CPU to a different thread. The operating system's *scheduler* is responsible for choosing which thread gets to run next. Although some operating systems such as Linux use highly complex scheduling algorithms, Embedded Xinu's scheduler uses a simple algorithm that simply dequeues and runs the next available thread from a data structure called the "ready queue", which contains the list of threads currently ready to run, sorted by thread priority.

A new thread is programatically created by calling `create()`, which is declared as follows:

```
tid_typ create(void *procaddr, uint ssize, int priority,
               const char *name, int nargs, ...);
```

`create()` takes in the address of the procedure to execute in the new thread, the stack size, the priority, the name of the thread, and any arguments to pass to the thread's start procedure. The return value is a unique thread identifier (with integer type `tid_typ`) for the newly created thread, or `SYSERR` if the new thread could not be created. Thread identifiers are actually indices into `thrtab`, which is a table of thread entries with a fixed size defined at compilation time.

A newly `create()`d thread begins in a suspended state and is not eligible to be run until `ready()` has been called to insert it into the ready queue:

```
int ready(tid_typ tid, bool resch);
```

A thread, identified by its thread ID, can be terminated using `kill()`, declared as follows:

```
syscall kill(tid_typ tid);
```

`kill()` frees the memory allocated for the thread's stack and marks the thread entry as free. A thread automatically `kill()`s itself if it reaches the end of the procedure with which it was started.

[3]One student has ported Embedded Xinu to the highly parallel Intel SCC platform [64]. However, Intel has since discontinued the platform, and consequently the code is no longer included in the current version of Embedded Xinu.

To switch between threads, Embedded Xinu must save the state of the currently executing thread and restore the state of a different thread. A thread's state primarily contains its *context record*, which contains the saved values of CPU registers. The format of the thread context record is architecture-dependent because each architecture has a different set of CPU registers and calling convention. In any case, switching threads is referred to as *context switching* and is implemented using the `ctxsw()` procedure. `ctxsw()` is a very low-level procedure that must be implemented in assembly language and is not intended to be called directly by applications. It is somewhat unusual, as far as procedures go, in that it effectively returns in a thread other than the one from which it was called; the "real" return to the original thread only happens when that thread is switched back to by a later call to `ctxsw()`.

By default, at any time during a thread's execution, the CPU can be interrupted. Such interrupts can arise from the timer being used to implement preemptive multitasking as well as from other hardware devices that have been configured to generate interrupts, typically by a device driver. Therefore, care must be taken when multiple threads could be accessing the same data concurrently. Short blocks of code can be executed with interrupts disabled by making use of the `disable()` and `restore()` functions. However, to maintain a high level of performance and responsiveness, interrupts should not be disabled for longer blocks of code. For this situation Embedded Xinu provides semaphores with the standard semantics. A thread can create a semaphore with an initial count by calling `semcreate()`, wait on a semaphore by calling `wait()`, and signal a semaphore by calling `signal()`.

Embedded Xinu also provides an optional *mailbox* module that provides structured inter-thread communication via message passing through FIFO (First In, First Out) queues called *mailboxes*.

Although the thread execution model described here is much simpler than the models used in other operating systems such as Linux, it shares many essential features and is a good starting point for understanding operating systems.

2.1.3 Device model

From the perspective of an operating system, a *device* is a hardware component with which software can communicate. Examples include serial ports, network controllers, non-volatile memory, keyboards, and monitors.

In UNIX-like systems, devices are, at one level, usually represented as files on which common operations, such as reading and writing, can be performed. Both Embedded Xinu and the original Xinu follow a similar but simplified device model. At compilation time, a fixed table of devices is defined based on a platform-specific configuration file. Each entry in the device table, or `devtab`, is of the format shown in Figure 1. Each entry contains a number of function pointers, each of which points to the implementation of the corresponding operation on the specific device. These function pointers are not accessed directly, but rather accessed through generic system calls such as `read()` and `write()`. In object-oriented terms, this is an example of

```
typedef struct dentry
{
    int     num;
    int     minor;
    char    *name;
    devcall (*init)(struct dentry *);
    devcall (*open)(struct dentry *, ...);
    devcall (*close)(struct dentry *);
    devcall (*read)(struct dentry *, void *, uint);
    devcall (*write)(struct dentry *, const void *,
       uint);
    devcall (*seek)(struct dentry *, long);
    devcall (*getc)(struct dentry *);
    devcall (*putc)(struct dentry *, char);
    devcall (*control)(struct dentry *, int, long,
       long);
    void *csr;
    void (*intr)(void);
    uchar irq;
} device;
```

Figure 1: Declaration of device table entry in Embedded Xinu. Note: "dentry" here stands for "device entry", not "directory entry" as it does in Linux.

polymorphism. However, not all operations make sense for all devices; for example, it makes no sense to `write()` to a keyboard. In such cases, the operation is redirected to a dummy function that simply returns `SYSERR` (the generic error code in Embedded Xinu).

The source code for Embedded Xinu's device drivers is located in subdirectories of the `device/` directory. Each device driver typically provides the ability to manage one or more devices of the same type (e.g. multiple network interfaces of the same model). Each such device is given its own device entry in the device table, but each device of a given type has a distinct minor number.

Embedded Xinu's device model can also represent *pseudo-devices* that are created by software and do not represent specific hardware devices. Examples of this are the TTY driver, which wraps around another device driver to provide capabilities such as line buffering suited for interactive use, and the UDP and TCP drivers, which manage the allocation of UDP and TCP devices (similar to sockets in other operating systems).

Since the Xinu device model is fairly simple, it does have some limitations. It only supports static devices, which places it at odds with buses such as USB that support dynamic addition and removal of devices. The same applies to UDP and TCP devices, which are usually allocated and freed at runtime and are not well-suited to being derived from static entries in the main device table.

2.1.4 Standard C library

To support applications and drivers, Embedded Xinu provides a set of *system calls*[4] which allow access to resources such as threads, semaphores, devices, and memory. On top of these raw system calls, it is very useful to have a set of standard utility functions to make programming easier. As such, Embedded Xinu implements a portion of a "standard C library" — that is, some of the functions that the C99 standard [63] specifies must be available in a conforming C implementation. These are located in `lib/libxc/` and include formatted input/output functions such as `scanf()` and `printf()`, string functions such as `strcmp()` and `strchr()`, memory management functions such as `malloc()` and `free()`, and a few miscellaneous functions such as `qsort()`. Since it is part of an instructional operating system, the C library focuses on simplicity and correctness rather than performance and full compliance with the C99 standard.

During my work to port Embedded Xinu to the Raspberry Pi, I discovered that the C library contained many bugs. Furthermore, most functions were poorly documented, and some were excessively complicated or confusing. Consequently, I rewrote and re-documented most of the library to ensure a high-quality, well-documented implementation.

2.1.5 Shell

When a user starts Embedded Xinu and connects to it using a platform-appropriate method such as a serial cable, the first thing they see is the Xinu Shell (xsh), which is the interactive command-line interface to the operating system. Figure 2 shows example usage of the shell.

New shell commands can be implemented fairly easily. However, for convenience, a special shell command **test** is provided that by default does nothing. Experimental code can be written in the corresponding file (`shell/xsh_test.c`) and will be executed when the **test** command is run.

2.1.6 Networking subsystem

Networking is a major component of Embedded Xinu that is available on most supported platforms. The networking subsystem supports various protocols and is designed to be easy to customize in educational settings [19].

Computer networking deals with the sending and receiving of packets, which typically use a series of nested protocols. In Embedded Xinu, incoming network packets are received by a network interface driver [5] and are passed up the networking

[4]Unlike in many operating systems, making a "system call" in Embedded Xinu does not actually cause a transition between processor modes.

[5]Currently, only Ethernet is supported; there is no support for wireless on any platform, despite the use of the Linksys WRT54GL, which is in fact a wireless router

```
(Embedded Xinu) (arm-rpi) #0 (e@zzz) Tue Nov 19 20:42:48 CST 2013

 134217728 bytes physical memory.
     32768 bytes reserved system area.
   3772032 bytes Xinu code.
      8192 bytes stack space.
 130404736 bytes heap space.


-------------------------------------------------------
       _____  ___.__                  .___   .__
       \   \/  /|__| _____  __ __     |  _ \ |__|
        \     / |  |/     \|  |  \    | |_| ||  |
        /     \ |  |   |  \  |  /    |  __/ |  |
       /___/\  \|__|___|  /____/     | |    |__|
             \_/        \/           |/          v3.14
-------------------------------------------------------

Welcome to the wonderful world of Xinu!
xsh$ ps
TID NAME             STATE PRIO PPID STACK BASE STACK PTR   STACK LEN
--- ---------------- ----- ---- ---- ---------- ----------  ---------
  0 prnull           ready    0    0 0x003A0E80 0x003A2DCC       8192
  1 USB scheduler    wait    60    0 0x07FFFFFC 0x07FFFF64       4096
  2 USB hub thread   wait    60    0 0x07FFEFFC 0x07FFEF6C       8192
  3 arpDaemon        wait    30    0 0x07FFCFFC 0x07FFCF74       4096
  4 rtDaemon         wait    30    0 0x07FFBFFC 0x07FFBF74       4096
  5 icmpDaemon       wait    30    0 0x07FFAFFC 0x07FFAF74       4096
  6 tcpTimer         sleep   20    0 0x07FF9FFC 0x07FF9F74      65536
  8 USB defer xfer   sleep  100    1 0x07FD9FFC 0x07FD9F7C       4096
  9 SHELL0           recv    20    7 0x07FD8FFC 0x07FD8D7C      65536
 10 SHELL1           wait    20    7 0x07FC8FFC 0x07FC8D24      65536
 19 ps               curr    20    9 0x07FE9FFC 0x07FE9E24       8192
xsh$
```

Figure 2: Example use of the Xinu Shell: the operating system printed an artistic logo and version number specific to the Raspberry Pi port after starting. Following that, the user ran the **ps** command, which displayed a table of threads in the system. The shell is now waiting for the next command to be entered. Threads are described in more detail in Section 2.1.2. A list of all shell commands supported by the build of Embedded Xinu can be obtained by running **help** with no arguments.

stack, often to the IPv4 module, then to the TCP or UDP driver, then to an application. Other paths are possible; for example, ARP and ICMP replies are generated directly by the networking subsystem, and the routing module can route IPv4 packets destined for elsewhere. When an application sends packets, they traverse the networking stack in the opposite direction and ultimately are sent out onto the network by the network interface driver.

The networking subsystem supports up to a fixed number of network interfaces (chosen at compilation time), each of which is associated with a fixed number of receiver threads that read networking packets from the interface. Each network interface corresponds to a distinct Ethernet port available on the underlying hardware. However, some platforms, such as the Raspberry Pi, only have one network interface available, and some do not support networking at all.

Various shell commands, such as **netup**, **netdown**, **route**, **ping**, **arp**, and **netstat**, let users interact with the networking subsystem. Example usage of **netup**, **route**, and **ping** is shown in Figure 3.

2.2 Testsuite

In contrast with the original Xinu operating system, Embedded Xinu follows modern software development practice by integrating testing into the development process. Embedded Xinu includes an optional runtime test suite that automatically tests many operating system features. It can be run by executing the **testsuite** command at the interactive shell. As shown by its output in Figure 4, the test suite tests a range of features. Regressions can be easily detected. Still, Embedded Xinu's test suite, like any other in any software project, is incomplete and cannot fully test all features. Furthermore, I discovered and fixed bugs in the test suite itself, which was frustrating.

2.3 Documentation

Since Embedded Xinu is an instructional operating system, it is critical that it be well-documented. Although the code in the original Xinu operating system was fairly straightforward, the main formal documentation seems to have been in the form of a book by Dr. Douglas Comer [22] that walked the reader through the operating system. This is certainly a good approach, but Embedded Xinu is a somewhat larger project and needs formal documentation of its own, ideally documentation that is easily accessible without requiring users to find or purchase a traditional book. Currently, Embedded Xinu's documentation can be divided into high-level documentation of the operating system as a whole and the API documentation.

Prior to my work, the high-level documentation consisted of articles on the Embedded Xinu Wiki [5], primarily written by former students at Marquette University. The articles covered operating system features, hardware information, and course curricula

```
xsh$ netup ETH0 192.168.0.2 255.255.255.0
WARNING: defaulting to no gateway
ETH0 is 192.168.0.2 with netmask 255.255.255.0 (no gateway)
xsh$ route
Destination     Gateway         Mask            Interface
192.168.0.0     *               255.255.255.0   ETH0
xsh$ ping 192.168.0.1
PING 192.168.0.1
52 bytes from 192.168.0.1: icmp_seq=0 ttl=64 time=1.112 ms
52 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.392 ms
52 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=0.379 ms
52 bytes from 192.168.0.1: icmp_seq=3 ttl=64 time=0.368 ms
52 bytes from 192.168.0.1: icmp_seq=4 ttl=64 time=0.299 ms
52 bytes from 192.168.0.1: icmp_seq=5 ttl=64 time=0.325 ms
52 bytes from 192.168.0.1: icmp_seq=6 ttl=64 time=0.373 ms
52 bytes from 192.168.0.1: icmp_seq=7 ttl=64 time=0.338 ms
52 bytes from 192.168.0.1: icmp_seq=8 ttl=64 time=0.403 ms
52 bytes from 192.168.0.1: icmp_seq=9 ttl=64 time=0.293 ms
--- 192.168.0.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9001ms
rtt min/avg/max = 0.293/0.428/1.112 ms
```

Figure 3: Example use of network shell commands. The user ran **netup** to bring up the network interface `ETH0` (the first Ethernet interface) with a static IP address of 192.168.0.2 and a netmask of 255.255.255.0. As shown by the output from the following **route** command, the netmask tells the routing module that packets destined for any IP address between 192.168.0.0 and 192.168.0.255 (inclusively) should be sent over `ETH0`. Finally, the user ran the **ping** command to test the connection with another computer on the network.

```
xsh$ testsuite
Test Suite  1: Argument Passing                          [PASS]
Test Suite  2: Priority Scheduling                       [PASS]
Test Suite  3: Thread Preemption                         [PASS]
Test Suite  4: Recursion                                 [PASS]
Test Suite  5: Single Semaphore                          [PASS]
Test Suite  6: Multiple Semaphores                       [PASS]
Test Suite  7: Counting Semaphores                       [PASS]
Test Suite  8: Killing Semaphores                        [PASS]
Test Suite  9: Process Queues                            [PASS]
Test Suite 10: Delta Queues                              [PASS]
Test Suite 11: Standard Input/Output                     [PASS]
Test Suite 12: TTY Driver                                [PASS]
Test Suite 13: Character Types                           [PASS]
Test Suite 14: String Library                            [PASS]
Test Suite 15: Standard Library                          [PASS]
Test Suite 16: Type Limits                               [PASS]
Test Suite 17: Memory                                    [PASS]
Test Suite 18: Buffer Pool                               [PASS]
Test Suite 19: NVRAM                                     [SKIP]
Test Suite 20: System                                    [PASS]
Test Suite 21: Message Passing                           [PASS]
Test Suite 22: Mailbox                                   [PASS]
Test Suite 23: Ethernet Driver                           [PASS]
Test Suite 24: Ethernet Loopback Driver                  [PASS]
Test Suite 25: Network Addresses                         [PASS]
Test Suite 26: Network Interface                         [PASS]
Test Suite 27: ARP                                       [PASS]
Test Suite 28: Snoop                                     [PASS]
Test Suite 29: UDP Sockets                               [PASS]
Test Suite 30: Raw Sockets                               [PASS]
Test Suite 31: IP                                        [PASS]
Test Suite 32: User Memory                               [PASS]
Test Suite 33: Simple TLB                                [SKIP]
```

Figure 4: Output from a sample run of the Embedded Xinu test suite on the Raspberry Pi. Tests 19 and 33 were automatically skipped because the corresponding features are not supported in the Raspberry Pi port.

and assignments. Many pages were out of date or unclear to outsiders, and much important information was missing. After a discussion with another developer on the Xinu mailing list, we decided to export the Wiki into reStructuredText (rst) documentation and distribute it directly with the source code, with HTML and PDF versions available online. Following this, I have been working to update the documentation, including adding information about the Raspberry Pi port as well as improving the documentation about existing operating system features. As of this writing the HTML documentation may be read at `http://embedded-xinu.readthedocs.org/en/latest/`. As of this writing the Wiki is still available, but it may soon be redirected to the new documentation.

The API documentation currently consists of documentation, primarily for function behavior, that is automatically generated from comments in the source code using Doxygen [61]. During my work with Embedded Xinu, I found that many functions were documented incorrectly or incompletely, so I have worked to improve this. As a result it should be easier for new programmers to understand and work with the source code.

2.4 Multi-platform support

Although Embedded Xinu was originally written for just one platform (the Linksys WRT54GL router), it has since been ported to other platforms, such as other MIPS-based routers and the Raspberry Pi. Consequently, like in many other operating systems, some subroutines in Embedded Xinu are implemented differently depending on the platform.

Platform-dependent code is organized by placing it in subdirectories of `arch/` or `platforms/` directories. Each platform has its own subdirectory in each `platforms/` directory, but multiple platforms that share the same underlying CPU architecture, such as ARM, can share the files in the appropriate `arch/` subdirectory.[6] `platforms/` and `arch/` directories can be found in three places:

- `loader/`, for architecture or platform-specific startup code;
- `system/`, for architecture or platform-specific runtime code such as interrupt handling, thread context setup, and context switching;
- `compile/`, for architecture or platform-specific configuration files, such as the file describing the static device table (see Section 2.1.3).

The list of currently supported platforms can be found in the documentation distributed with Embedded Xinu.

[6] The `arch/` directories are something I added. They reduce code duplication among multiple MIPS and ARM platforms and make it easier to port Embedded Xinu to new platforms with these CPU architectures.

2.5 Development, code access, and compiling

Previously, Embedded Xinu was developed with the help of the Subversion version control system, using a private Subversion repository hosted by a server at Marquette University. However, to open Embedded Xinu development to more developers and move to a more advanced, distributed version control system, I (in coordination with previous students who had worked on Embedded Xinu) moved the sources over to the Git version control system, hosted in a repository publicly available on Github at `https://github.com/xinu-os/xinu`. As a result, is now easy for anyone to obtain the latest development sources of Embedded Xinu and optionally modify the code for their own purposes. It is also now easier for developers to work on temporary or experimental features without interfering with the main code, since Git features easy branching and merging.

The details of using Git are outside the scope of this thesis. Many resources are freely available online, and much documentation is distributed with the Git software itself. However, once Git has been installed on a UNIX or Linux system, the Embedded Xinu repository can be cloned into the `./xinu/` directory by running the following command:

```
$ git clone https://github.com/xinu-os/xinu
```

Downloads of stable versions of Embedded Xinu are still available on the old website at `http://xinu-os.org/Downloads`, but as of this writing there is no stable version that supports the Raspberry Pi.

Building Embedded Xinu requires several de-facto standard tools, such as a GCC cross compiler targeting the appropriate CPU architecture and the `make` utility.[7] A *cross compiler* refers to a compiler that targets a CPU architecture that is not the same as the CPU architecture on which the compiler runs.[8] The Raspberry Pi uses an ARM processor, so compiling XinuPi requires a GCC cross compiler targeting the ARM architecture, specifically the `arm-none-eabi` target.[9]

Once the needed tools have been installed, Embedded Xinu can be compiled by running a command similar to the following:

```
$ make -C compile/ PLATFORM=arm-rpi
```

The `-C compile/` option tells the `make` utility to process the Makefile in the `compile/` directory rather than the current directory. The `PLATFORM=arm-rpi`

[7] The build process also assumes a UNIX-like host system, so Microsoft Windows can only be used if Cygwin is installed.

[8] Although some compilers, such as clang, can natively produce code for multiple CPU architectures, a given installation of GCC can only produce code for one CPU architecture.

[9] `arm` specifies the ARM architecture, whereas `none-eabi` specifies that the compiler shall produce bare-metal code compatible with the ARM Embedded Application Binary Interface, part of which is the ARM Procedure Call Standard[12].

option sets the platform for which to compile Embedded Xinu. `arm-rpi` stands for *ARM (Raspberry Pi)*; another possibility is `wrt54gl` (*Linksys WRT54GL*).

The compilation will produce the file(s) needed to boot Embedded Xinu on the target platform. As an example, for the Raspberry Pi this will be the file `compile/xinu.boot`, which will be a raw binary image containing the kernel's code and data. This file must be copied to `kernel.img` on the FAT32 partition of the memory card inserted into the Raspberry Pi.[10]

More details about compiling Embedded Xinu, including acquiring or building an appropriate cross-compiler, can be found in the official documentation, especially the *Getting Started* page, which I have rewritten.

[10] Although the Raspberry Pi can only boot from memory card, indirect ways to start XinuPi do exist; for example, see Chapter 8.

Chapter 3

The Raspberry Pi

3.1 Motivation and history

Despite the increasing availability of personal computers, many students entering college today, at least in some places, do not seem have more programming experience than typical students in the 1990s did, other than possibly in web programming [42]. Various factors, such as the high prevalence of point-and-click interfaces and the increasing complexity of modern hardware may be contributing to this trend. The Raspberry Pi is primarily an attempt to get more kids interested in programming and revive the hobbyist community by offering a low-cost, convenient, and well-supported platform for programming, or anything else a Linux system might be used for.

The Raspberry Pi was designed and is being sold by the Raspberry Pi Foundation, a registered U.K. charity [42]. Two models are available: the Model A, which costs $25, and the Model B, which costs $35. The first units were produced in early 2012, and since then over 2.5 million have been sold as of March 2014 [53]. Production started in China but has since moved to the U.K [55]. The Raspberry Pi is sold online.

To maintain its low cost, the Raspberry Pi takes advantage of low-cost hardware often used in modern embedded systems. Specifically, it uses a system-on-a-chip that integrates a CPU (Central Processing Unit), GPU (Graphics Processing Unit), RAM (Random Access Memory), and various peripheral devices into a single low-power chip in the center of the board. Furthermore, the Raspberry Pi itself is just a board, and buying one may or may not (depending on the package) include hardware or accessories necessary for actually using it. These accessories include a microUSB charger, a monitor, a mouse, and a keyboard, all of which are assumed to be already available to the user, bought separately, or else bought in a convenience package with a Raspberry Pi board.

Although the Raspberry Pi was originally designed for kids [42], it has attracted a strong hobbyist community. Hobbyists use the Raspberry Pi for various purposes such as automation, media centers, servers, gaming, electronics projects, and programming. There is an increasing number of formal publications regarding the Raspberry Pi, but

the largest amount of information can be found on the official Raspberry Pi forums [7]. A full discussion of the uses of the Raspberry Pi is beyond the scope of this thesis, which focuses on "bare-metal" programming in the context of porting an operating system.

The Raspberry Pi is primarily intended to run a Linux-based operating system, which is a good fit for high-level programming and provides useful experience with a UNIX-like operating system. Other operating systems such as FreeBSD [58], Plan 9 [57], and RISC OS [51] have also been ported to the Raspberry Pi. However, none of these operating systems provides the systems-level educational focus that Embedded Xinu does.

3.2 Hardware

3.2.1 Board overview

The Raspberry Pi is a credit-card-sized circuit board which measures 85.60mm $\times$ 56mm $\times$ 21mm and has a mass of 45g [40]. Schematics for various models and revisions of the circuit board have been published by the Raspberry Pi Foundation [54]. At the center of all versions of the board is the BCM2835 System-on-a Chip (SoC), which contains the CPU, GPU, memory, and various peripherals. Around the edges of the board are various connectors, such as GPIO pins, RCA video, HDMI, MicroUSB (power only; data pins are not connected), 3.5mm TRS Audio, two USB ports (only one on the Model A), a memory card slot, and an Ethernet port (Model B only). Figure 5 shows a diagram of the major components of the Raspberry Pi.

The Raspberry Pi has no power switch, "on button", or "off button". To turn it on, the user simply must plug in a power source. To turn it off, the user simply must unplug the power source. The Pi can be powered through the MicroUSB port (recommended by the Raspberry Pi Foundation), through the GPIO pins, or through USB port back-power. The Pi requires 300–700 mA at 5V, depending on several factors such as the USB devices being powered. The Ethernet controller is a USB device and consumes a significant amount of power if it is enabled on the Raspberry Pi Model B [6].

3.2.2 BCM2835 SoC

The BCM2835 SoC — the Broadcom 2835 System-on-a-Chip — contains the most important components of the Raspberry Pi, such as the CPU, GPU, and memory.

The CPU on the BCM2835 is an ARM-architecture processor, specifically an ARM1176JZF-S processor of the ARMv6 architecture revision. ARM processors are extremely widely used in electronic devices, including most cell phones, and are known for being inexpensive and power-efficient. The ARM architecture is extensively documented by ARM Ltd. in the *ARM Architecture Reference Manual* [10] as well as in reference manuals for specific ARM implementations such as ARM1176JZF-S [11].

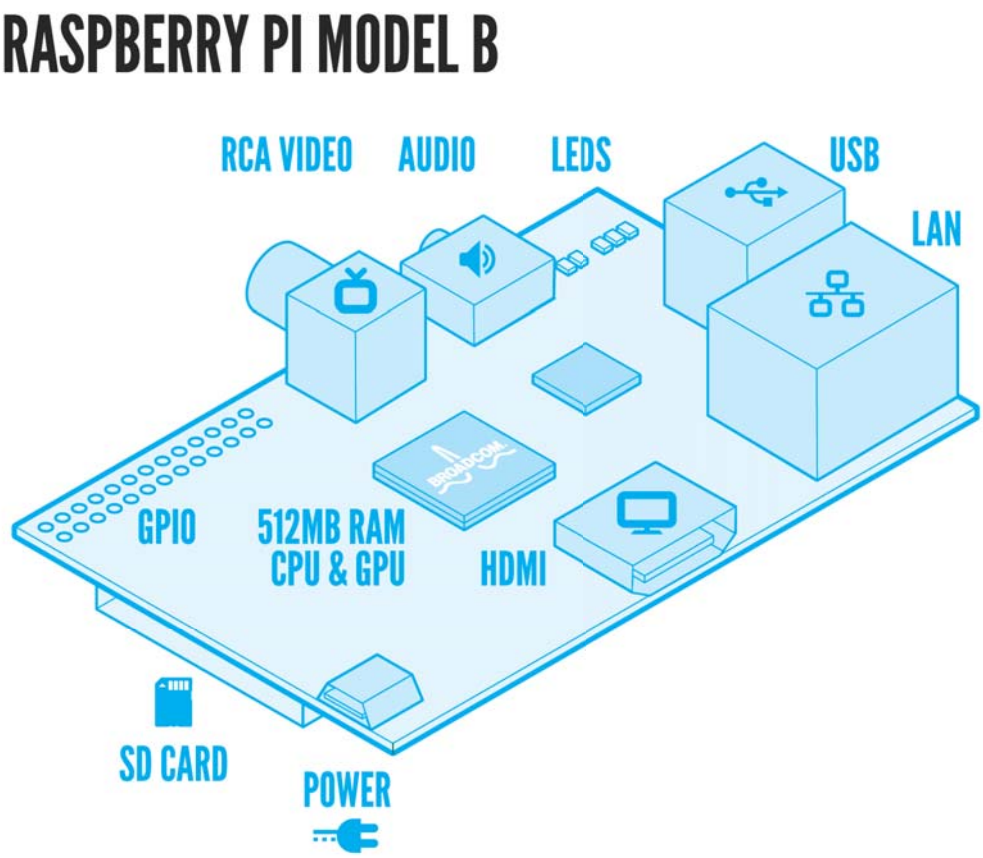

Figure 5: Diagram of the Raspberry Pi Model B. Image credit: Raspberry Pi Foundation; Creative Commons attribution and share-alike [41]. The chip labeled "512MB RAM, CPU & GPU" is the BCM2835. In real life it does not actually say "Broadcom" on it, since the chip actually consists of a memory package, provided by a different company such as Hynix or Samsung, physically stacked on top of a Broadcom package using package-on-package technology. The unlabeled square chip above and right of the BCM2835 is the SMSC LAN9512 USB 2.0 Hub and Ethernet Controller.

Since ARM processors are based on a RISC (Reduced Instruction Set Computing) design, programming for them at the assembly language level is slightly more straightforward than for other microprocessor architectures such as x86. However, the ARM architecture has still evolved a significant amount of complexity.

The GPU on the BCM2835 is a Broadcom VideoCore IV. Compared to the CPU, little is known about it because it is a proprietary Broadcom design for which no official documentation exists for end users.[1] Still, the GPU is made available to Linux for displaying graphics, and also video decoding if the user buys the appropriate license keys. According to a project which has attempted to reverse engineer aspects of the VideoCore, the VideoCore actually contains its own dual-core CPU which can run scalar and vector programs, as well as an image sensor pipeline and "Quad-Processor Units" which provide 24 GFLOPS compute performance for coordinate, vertex, and pixel shaders [28]. The VideoCore is believed to run its own operating system called VideoCore OS that is based on a proprietary real-time operating system called ThreadX, and one project has succeeded in reverse engineering the hardware enough to run custom code on it [29]. Additional information about the VideoCore may be gleaned from Linux code that interacts with it and from Broadcom U.S. patents dealing with an unnamed multimedia processor that may be the VideoCore [8].

Luckily, incomplete knowledge of the VideoCore is not very consequential for Embedded Xinu because the VideoCore is largely out of the picture when Xinu is running on the ARM processor. It is only really needed for graphics. (See Section 9.2 for more details.)

The BCM2835 also includes RAM (random access memory) and various peripheral devices. The amount of RAM was originally 256 MiB on all models of the Raspberry Pi but has been increased to 512 MiB on the Raspberry Pi Model B [56]. Some details about peripheral devices, such as the PL011 UART, system timer, interrupt controller, and USB controller, can be found in a datasheet[16] released by Broadcom. However, many peripheral devices are not well documented.

3.2.3 Memory card

The Raspberry Pi contains a slot into which the user can insert a memory card which may be either an SD (Secure Digital) card or MMC (MultiMediaCard). Although a purchase of a Raspberry Pi does not necessarily include a memory card (to keep costs down), the designers intended that an inserted memory card provide the Raspberry Pi's main nonvolatile storage. In addition, an inserted memory card is essential for the Raspberry Pi's boot process.

[1] This paragraph expresses the historical view and does not account for Broadcom's very recent (February 28, 2014) release of register-level documentation for the VideoCore IV graphics engine and source code for the VideoCore IV 3D graphics subsystem used in the BCM21553 3G integrated baseband SoC [52]. Analyzing this release would not be straightfoward, partly because it does not, in fact, include the source for the `start.elf` blob that actually runs on the BCM2835. However, this documentation and source release is seemingly a huge step forward for better understanding of the VideoCore.

Software can read from, write to, or detect a memory card by using the Arasan External Mass Media Controller exposed in the peripheral address range of the BCM2835. The Arasan controller mostly, but not completely, conforms to the *SD Host Controller Specification*[50]. In addition, the Arasan controller is partially documented by Broadcom [16], but the original Arasan documentation does not appear to be publicly available.

3.2.4 USB and networking

Like many modern computer systems, the Raspberry Pi relies heavily on USB (Universal Serial Bus) to attach various devices, such as keyboards and mice. The Raspberry Pi Model B has two USB ports and one Ethernet port, whereas the Raspberry Pi Model A has one USB port and no Ethernet port. When running Linux, both Raspberry Pi models are compatible with a wide range of USB devices.

At the hardware level, both Raspberry Pi models support USB through the "Synopsys DesignWare Hi-Speed USB On-the-Go Controller" included in the BCM2835 [16]. This USB controller has no publicly available documentation, which has impeded efforts to write software that uses it; however, in Chapter 5, I describe my efforts to do exactly this.

Regardless of the status with documentation, it is known that the USB controller is associated with a single "host port" that represents a port on the USB root hub — that is, a link to the root of the tree of USB devices that are located on the Universal Serial Bus at any given time.[2] On the Model A, the host port is connected directly to the USB port accessible to the user, whereas on the Model B the host port is internally connected to a SMSC LAN9512 USB 2.0 Hub and Ethernet Controller. The internal hub of the SMSC LAN9512 then has three USB ports, two of which correspond to the externally visible USB ports on the Model B and one of which is used internally for the Ethernet device itself. In Chapter 6, I describe my efforts to write a driver that controls this Ethernet device.

3.2.5 Booting

When power is applied, the Raspberry Pi initially runs firmware on the VideoCore coprocessor. The firmware loads two auxiliary files, the main operating system kernel, and an optional configuration file from the inserted memory card. (The "main" operating system here refers to the operating system, such as Linux or Embedded Xinu, that gets started on the ARM processor, as opposed to Broadcom's proprietary code that runs on the VideoCore.)

More specifically, the memory card must contain an MS-DOS-style partition table with at least one FAT32-formatted partition containing the necessary boot files

[2] See Section 5.2.1 for information about USB topology in general.

in its root directory. The boot files include two[3] files provided by Broadcom — `bootcode.bin` and `start.elf` — along with the user-provided `kernel.img` [44]. `bootcode.bin` and `start.elf` are both binary files that cannot easily be modified or disassembled because they contain code that uses one of the VideoCore's undocumented instruction sets [2]. Furthermore, according to Broadcom's license file, these files may be redistributed but not modified [27]. The user-provided file `kernel.img`, on the other hand, is ARM code that the firmware loads and starts at memory address `0x8000`. In addition, the user may provide an optional file `config.txt` to customize certain boot parameters, such as the physical memory split between the ARM and VideoCore [21].

`bootcode.bin`, `start.elf`, and other VideoCore-related files may be downloaded from a Git repository maintained by the Raspberry Pi Foundation [27].

3.3 Documentation

Various guides, tutorials, blogs, and even published books are readily available for users of the Raspberry Pi. However, these resources almost always assume the Raspberry Pi runs a Linux-based operating system. As hinted at by Section 3.2.2, Section 3.2.3, and Section 3.2.4, lower-level details about the Raspberry Pi hardware are sometimes not readily available. Basic information about the ARM-accessible peripherals can be found in a datasheet released by Broadcom [16]. However, other details — for example, most information about the VideoCore — are undocumented. In some cases, the best information exists in forum posts by Raspberry Pi Foundation members who are familiar with the board's design and have access to non-public documentation from Broadcom, Synopsys, and/or Arasan [21][2][1][3]. In other cases, the best information exists in the logic contained in the C source code of Linux drivers [49][43][33].

Consequently, this thesis attempts to clarify some aspects of the Raspberry Pi hardware that are not well documented elsewhere. However, due to space limitations, lack of information, and the high complexity of certain devices, it still cannot fully document the Raspberry Pi hardware.

[3]Originally, a third file, `loader.bin`, was required as well. However, with the newest firmware files, `loader.bin` is no longer required.

Chapter 4

Porting the Core Operating System

This chapter discusses the work that was necessary to get a basic Embedded Xinu port running on the Raspberry Pi. The scope of this chapter includes the early booting of the operating system, functional preemptive multitasking, and support for basic textual input/output over a serial connection. USB support and networking support are not considered here (see chapters 5 and 6).

I was the primary contributor to this portion of the XinuPi project. However, Tyler and Farzeen also made contributions, and some parts were based on prior work on Embedded Xinu.

4.1 Loader

In Embedded Xinu, the *loader* is assembly language code responsible for very early initialization of the system. The loader begins with the first instruction executed in the kernel and ends when control is passed to `nulluser()`, the platform-independent C startup code. Note that by this definition, `nulluser()`, which is primarily responsible for initializing device drivers and platform-independent operating system structures before entering an infinite loop, is *not* considered part of the loader. Many operating systems share this same general design; for example, Linux begins by running platform-dependent startup code before passing control to the `start_kernel()` C procedure. Embedded Xinu's loader differs in being simpler, since its only goal is to set up a usable C execution environment as fast as possible.

Tyler Much, Farzeen Harunani, and I implemented the loader for XinuPi. It is located in `loader/platforms/arm-rpi/start.S`. This assembly language file is compiled into an object file and linked into the very beginning of the kernel image at address `0x8000`, which is the address at which the Raspberry Pi firmware loads the kernel image. As a result, when the firmware passes control to XinuPi, execution begins at the first assembly language statement in `start.S`.

The first major task of XinuPi's loader is to set up the ARM exception vector table by copying it from a location in the kernel image to memory address 0. This table

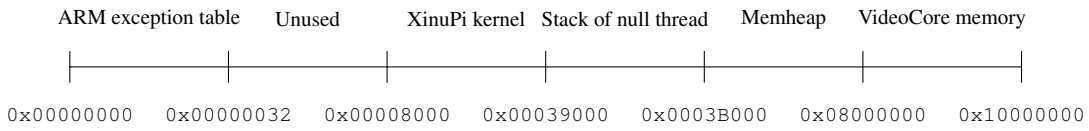

Figure 6: Example physical memory map (RAM only, not to scale) when XinuPi is running on the Raspberry Pi.

associates code with different ARM exceptions, and I will discuss it in more detail in Section 4.2. Figure 6 shows this table in its final location.

The next major task of XinuPi's loader is to zero the `.bss` section of the in-memory kernel image. The `.bss` section contains C variables that were declared with static storage duration but not explicitly initialized. According to section 6.7.8.10 of the C99 standard [63], such variables must be initialized to zero before a C program actually starts running.[1] A C compiler could implement this behavior by explicitly including the zeroes in the resulting object file. However, modern compilers such as GCC save space in the resulting object file by not explicitly allocating these variables and instead placing them in the `.bss` section, which is expected to be expanded and zeroed at runtime. In XinuPi, the loader is responsible for this task.

The last major task of XinuPi's loader is to reserve memory for operating system use. All memory after the end of the kernel image up to the region reserved by the firmware for the VideoCore is available. Out of this available memory, XinuPi's loader sets aside a small chunk for the stack of the "null thread", which runs the `nulluser()` C startup code and becomes the first thread. The loader stores the memory address of this stack in `sp`, the stack pointer register of the ARM processor, before entering `nulluser()`. This is necessary because C code must have a stack available to execute.[2] However, before doing this, XinuPi's loader reserves the remaining memory for the *memheap*, which is the region of memory that Embedded Xinu uses for dynamic memory allocations.

Implementing the loader required some degree of familiarity with ARM processors, including the exception handling model and the assembly language. ARM Ltd. provides this information in their extensive documentation for the ARM architecture in general [10] as well as for specific ARM implementations [11]. In addition, non-official information about ARM programming is widely available online due to the widespread use of ARM processors in embedded systems.

[1] Actually, pointer variables must be initialized to the implementation-defined null pointer representation, but this happens to be all zero bits on virtually all modern implementations.

[2] Technically, this is not a requirement of the C language itself, but rather a characteristic of the specific implementation provided by GCC compiling for ARM processors. However, the use of a stack is ubiquitous across modern CPU architectures.

4.2 Interrupt handling

Interrupt handling is an essential part of almost all modern operating systems. At the hardware level, an interrupt is an electrical signal that informs the CPU that something has happened that needs its attention — for example, the expiration of a timer or the completion of a read from or write to a hardware device.

On ARM architectures, interrupt handling can be divided into several areas of concern:

1. The standard ARM mechanism for handling exceptions;
2. The standard ARM mechanism for temporarily disabling interrupts;
3. A hardware implementation-defined mechanism for enabling or disabling interrupts from specific devices and determining which device(s) have interrupts pending when the generic IRQ exception is received;
4. Actually responding (in software) to an interrupt from a specific device.

The first subsection below describes (1) and (2), the ARM-specific details. The next subsection describes (3), which on the Raspberry Pi is the BCM2835-specific details. Concern (4) is device-specific and is dealt with in other chapters and sections (e.g. Chapter 5 for USB, Section 4.3 for the hardware timer).

4.2.1 ARM interrupt handling

The ARM exception model is common to all ARM-based platforms and is not specific to the Raspberry Pi. Therefore, the discussion here (and the corresponding code) is also applicable to other ARM-based platforms to which Embedded Xinu might be ported in the future.

The ARM architecture defines a number of *exceptions*, each of which corresponds to a different type of asynchronous event. The two exceptions dealing with interrupts are the *IRQ (Interrupt Request) exception* and *FIQ (Fast Interrupt Request) exception* [10].

Normally, when an ARM processor receives an electrical interrupt signal, an IRQ exception is generated. Software or hardware can optionally prioritize interrupts from a specific device by assigning them to the FIQ exception rather than the IRQ exception. However, because the use of the FIQ exception is optional from the viewpoint of software running on the Raspberry Pi, for simplicity I have not used it in XinuPi and do not consider it further in this thesis.

As documented in Section A2.5.6 of the *ARM Architecture Reference Manual* [10], bit 7 of the ARM `cpsr` (Current Program Status Register) is 0 if and only if the IRQ exception can currently be received by software. If the ARM processor receives an IRQ exception while this bit is 1, the exception is delayed until this bit is changed to 0, at

which point the IRQ exception occurs immediately. Since this bit can be modified by software using the `msr` and `mrs` instructions[3], this bit provides a way to implement the `enable()`, `disable()`, and `restore()` procedures that are available in the ports of Embedded Xinu to other architectures, such as MIPS. In my implementation of these procedures, `enable()` sets bit 7 of the `cpsr` to 1, `disable()` sets bit 7 of the `cpsr` to 0 and returns the old value of the `cpsr`, and `restore()` restores bit 7 of the `cpsr` to the value specified in the provided mask which was obtained by a prior call by `disable()`. The `disable()` and `restore()` subroutines in particular are used in many places in Embedded Xinu to protect critical regions of code throughout which an interrupt cannot occur, otherwise a race condition would result. Such race conditions arise because Embedded Xinu is a preemptive multitasking operating system and the CPU can be rescheduled to another thread whenever the timer interrupt occurs.

When an ARM processor receives an exception that is not blocked by the settings in the `cpsr`, it enters an exception-specific mode and branches to one of a number of fixed memory addresses known as *exception vectors* [10]. By default, ARM processors expect that the table of exception vectors, with one entry per exception type, be located in consecutive words at physical memory address 0. Operating systems are expected to install an exception handler for every exception by writing an appropriate ARM instruction[4] to the corresponding exception vector. In XinuPi, this is done in the loader (see Section 4.1). However, XinuPi is currently only concerned with the IRQ exception, which will be discussed below.

In the IRQ exception vector, XinuPi's loader installs a pointer to a hand-coded ARM assembly procedure `irq_handler()`. This procedure, which I wrote, is shown in Figure 7. It is the top-level interrupt handling routine and is essentially a wrapper around the C procedure `dispatch()`. Excluding the `dispatch()` implementation, `irq_handler()` is not Raspberry Pi-specific and should be applicable to any ARM platform to which Embedded Xinu might be ported.

When the ARM processor has started processing an IRQ exception and has branched to `irq_handler()`, it has already automatically entered a special processor mode called *IRQ mode*. IRQ mode uses banked registers for `sp` (stack pointer) and `lr` (link register), which means that accesses to these named registers logically access a different location in IRQ mode than in other processor modes. The other general-purpose registers are not banked, which means that they still contain the values they contained when the processor received the IRQ exception. To avoid corrupting the state of the interrupted thread, `irq_handler()` must save the values of any non-banked registers that it needs to use. Because it calls the C function `dispatch()` using the ARM Procedure Call Standard [12], registers `r0` – `r3`, `r12`, and `lr` may be clobbered

[3] `msr` means "Move to system control coprocessor from ARM register" and `mrs` means "Move to ARM register from system control coprocessor". In this terminology, the `cpsr` is considered to be one of the system control coprocessors, whereas "ARM register" refers specifically to one of the general-purpose registers. The `cpsid` (Change Program State Interrupt Disable) instruction can also be used to modify the `cpsr`.

[4] ARM is a RISC architecture that uses a fixed-size instruction coding (4 bytes per instruction).

```
1.   irq_handler:
2.       sub    lr, lr, #4
3.       srsdb  #ARM_MODE_SYS!
4.       cpsid  if, #ARM_MODE_SYS
5.       push   {r0-r4, r12, lr}
6.       and    r4, sp, #4
7.       sub    sp, sp, r4
8.       bl     dmb
9.       bl     dispatch
10.      bl     dmb
11.      add    sp, sp, r4
12.      pop    {r0-r4, r12, lr}
13.      rfeia  sp!
```

Figure 7: The top-level interrupt handling routine for ARM ports of Embedded Xinu. See the original in `system/arch/arm/irq_handler.S` for the full comments; here I give only a brief walkthrough. Line 1 corrects the link register, which is necessary because of a quirk in how ARM processors enter the exception-handling code. Lines 2 and 3 save the return state on the IRQ mode stack and enter System mode. Line 5 saves any registers that may be clobbered by `dispatch()` on the System mode stack. Lines 6 and 7 ensure the stack pointer is 8-byte aligned before calling `dispatch()`. Lines 8 and 10 bracket the call to `dispatch()` with *data memory barriers*, which guarantee proper ordering of memory accesses to peripherals. Without them, it is theoretically possible for peripheral accesses of the interrupted thread and the interrupt handling code to be mixed up. Line 11 returns the stack pointer to its original alignment, line 12 restores saved registers, and line 13 returns control to the currently executing thread using the saved return state.

(i.e. their contents overwritten) and therefore must be saved and restored around the call to `dispatch()`.

An additional complication in `irq_handler()` arises from the fact that IRQ mode is designed to use a separate stack. Although this is a convenient feature that automatically prevents stack overflows that could be caused by executing the interrupt handler using the stack of the currently running thread, it also adds more complexity because it prevents straightforward rescheduling of a thread from an interrupt handler. Since Embedded Xinu is designed for simplicity, I decided the most straightforward solution would be to have `irq_handler()` change the processor mode back to the mode in which the processor normally operates. Thus, XinuPi runs interrupt handlers on the stack of the currently executing thread, and interrupt handling code must take care to use limited stack space.

More details about this aspect of interrupt handling can be found in the source code itself in the file `system/arch/arm/irq_handler.S`.

4.2.2 BCM2835 interrupt handling

The C subroutine `dispatch()` is responsible for passing control to more specific interrupt handling subroutines based on which IRQs are currently pending. (The usage of the word "IRQs" here refers to specific interrupt request lines, as opposed to the generic IRQ exception which occurs when any of these IRQs is received.) Each such IRQ corresponds to a specific device or hardware module that can issue interrupts. This portion of interrupt handling is specific to the BCM2835 SoC (System-on-a-Chip) used on the Raspberry Pi because it relies on a memory-mapped interrupt controller provided on the SoC. This interrupt controller is documented (but not particularly well explained) in Section 7 of Broadcom's *BCM2835 ARM Peripherals* datasheet [16]. I shall refer to this interrupt controller as the "BCM2835 ARM Interrupt Controller", with the "ARM" distinction being needed because, although not officially documented, the BCM2835 has a separate interrupt controller for the VideoCore [30].

ARM-accessible IRQs on the BCM2835 are of two types: ARM-specific IRQs and shared IRQs. The latter are shared between the ARM and VideoCore and can be received by either. A total of 8 ARM-specific IRQs and 64 shared IRQs are available. XinuPi maps these two types of IRQs into a single continuous range such that the 64 shared IRQs are numbered 0–63 and the ARM-specific IRQs are numbered 64–71. Table 1 shows the meanings of some of the important IRQs.

In the initial hardware state, all IRQs are disabled, so the ARM processor cannot receive any IRQ exceptions. To enable an IRQ, software must write 1 to the corresponding bit in one of the three Enable registers of the BCM2835 ARM Interrupt Controller (shown in Table 2). In XinuPi, this is implemented in `enable_irq()`, which takes in the number of the IRQ to enable. After doing this, the ARM processor will receive an IRQ exception when the corresponding device electrically asserts its interrupt request line. In XinuPi, this will eventually result in a call to `dispatch()`, which checks the Pending registers of the BCM2835 ARM Interrupt Controller (shown in Table 2) to

IRQ	Device	Notes
0	System Timer Compare Register 0	Do *not* enable this IRQ; it's already used by the VideoCore.
1	System Timer Compare Register 1	See Section 4.3.
2	System Timer Compare Register 2	Do *not* enable this IRQ; it's already used by the VideoCore.
3	System Timer Compare Register 3	See Section 4.3.
9	USB controller	This is the *only* USB IRQ because all communication with USB devices happens through the USB controller. See Section 5.3.4 for details.
55	PCM Audio	Mentioned in Section 9.1.
62	SD host controller	Mentioned in Section 3.2.3.

Table 1: Incomplete list of BCM2835 IRQ numbers, specifically the ones that have been tested as part of the XinuPi project. The full list of IRQ numbers is not officially documented, but it can be found declared in a Linux header [15]. In the table, I use the numbering scheme used by both Embedded Xinu and Linux, where the shared IRQs are numbered 0–63. The ARM-specific IRQs (none shown) would be numbered starting at 64.

determine which IRQs are pending. For each pending IRQ, `dispatch()` calls the corresponding interrupt handling function, which is typically part of a device driver. Each such interrupt handling function is responsible for clearing the source of the IRQ using a device-specific method. For example, when the timer interrupt occurs, the IRQ-specific interrupt handling function is `clkhandler()`, and it clears the IRQ by writing to a certain bit in the BCM2835 System Timer registers (see Section 4.3 for more details of this particular device).

More details about this aspect of interrupt handling can be found in the source code itself in the file `system/platforms/arm-rpi/dispatch.c`.

4.3 System timer

For any operating system to support preemptive multitasking, a hardware timer that can interrupt the CPU at programmable times must be available. On each platform, Embedded Xinu encapsulates all interaction with the hardware timer into two functions:

- `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 number of timer cycles elapsed since the timer last reset or overflowed.

On the Raspberry Pi, the BCM2835 System Timer can be used to implement these two functions. This timer, which is a memory-mapped peripheral, is partially

Offset	Name	Description
+0x200	IRQ_basic_pending	Bitmask of pending ARM-specific IRQs, as well as additional bits (not currently used by XinuPi) to accelerate interrupt handling
+0x204	IRQ_pending_1	Bitmask of pending shared IRQs 0–31
+0x208	IRQ_pending_2	Bitmask of pending shared IRQs 32–63
+0x20C	FIQ_control	An IRQ number can be written to this register to cause the FIQ exception to be issued instead of the IRQ exception; not currently used by XinuPi
+0x210	Enable_IRQs_1	Write 1 to the corresponding bit(s) to enable one or more shared IRQs in the range 0–31
+0x214	Enable_IRQs_2	Write 1 to the corresponding bit(s) to enable one or more shared IRQs in the range 32–63
+0x218	Enable_Basic_IRQs	Write 1 to the corresponding bit(s) to enable one or more ARM-specific IRQs
+0x21C	Disable_IRQs_1	Write 1 to the corresponding bit(s) to disable one or more shared IRQs in the range 0–31
+0x220	Disable_IRQs_2	Write 1 to the corresponding bit(s) to disable one or more shared IRQs in the range 32–63
+0x224	Disable_Basic_IRQs	Write 1 to the corresponding bit(s) to disable one or more ARM-specific IRQs

Table 2: BCM2835 ARM Interrupt Controller registers, located at physical memory address 0x2000B000. Note that the offsets start at 0x200, not 0x00. The purpose (if any) of the first 0x200 bytes is undocumented.

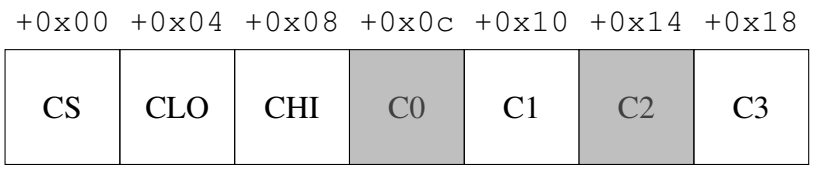

Figure 8: BCM2835 System Timer registers, based at physical memory address 0x20003000. CLO and CHI make up the 64-bit free running counter. C0 through C3 are the output compare registers, but only C1 and C3 are available to the ARM.

documented by Broadcom [16]. It has a 64-bit free-running counter, shown as `CLO` (Clock Low) and `CHI` (Clock High) in Figure 8, 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 (`CS`). In that register, software must set the bit corresponding to the output compare register index (0—3) to clear the corresponding timer interrupt.

A caveat not documented in [16] 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, could make it possible to implement a variety of embedded and real-time software for the Raspberry Pi. Furthermore, Broadcom's documentation describes an additional timer, the "ARM Timer", that is also available to software [16].

4.4 Creating and switching between threads

Operating systems supporting multiple threads must be able to create new threads as well as switch between threads. The low level details of these tasks must be implemented differently on each CPU architecture. Some prior work had been done on these details for potential ARM ports of Embedded Xinu. In particular, some students at the Rochester Institute of Technology have worked to port Embedded Xinu to several ARM platforms, including the Raspberry Pi [32]. However, I found that their code relevant to the thread implementation was not well explained and contained at least two bugs (the code did not correctly set the interrupt state of new threads, nor did it maintain the 8-byte stack alignment required by the ARM Procedure Call Standard [12]). Therefore, I ended up re-implementing the low-level details regarding multithreading. I also refactored the `create()` subroutine, which is responsible for creating new threads, to separate the platform-dependent portion into a new `setupStack()` subroutine. This reduced code duplication among the supported Embedded Xinu platforms.

More general information about Embedded Xinu's threading model can be found in Section 2.1.2. Here I discuss my implementations of `setupStack()` and `ctxsw()` for ARM platforms. Briefly, `setupStack()` is responsible for setting up the stack for a new thread (including setting initial CPU register values), and `ctxsw()` is responsible for switching between the stacks of two threads (including saving and restoring CPU register values). Both functions deal with an architecture-dependent structure called the *thread context record*, which contains the saved values of various CPU registers. Table 3 shows the format of this structure used in XinuPi; however, the same format would also be applicable to any other ARM-based platform to which Embedded Xinu might be ported.

The ARM Procedure Call Standard [12] defines some registers as *callee-save*, meaning that the called procedure is responsible for saving their values (as the caller expects their values to be preserved), and some registers as *caller-save*, meaning that the caller of the procedure is responsible for saving their values (as they are free to be clobbered by the called procedure). Here, all callee-save registers must be included in the thread context record because they may be clobbered by any thread and therefore must be saved and restored across a context switch. The caller-save registers `r0` – `r3` and `r12` need not be included in the context record because the compiler already assumes them to be clobbered by a call to `ctxsw()`; however, `r0` – `r3` are included anyway because they are used to pass up to the first four arguments to the starting procedure of a new thread. In addition, `cpsr` must be included in the context record to store the current IRQs disabled/enabled state, although this state is in fact only "enabled" for new threads because IRQs are disabled when performing a context-switch. Finally, the `sp` (stack pointer) register is not included in the context record because it must be restored before the context record can even be accessed; as such, each thread's stack pointer is saved in its entry in the global thread table (`thrtab`).

When `setupStack()` creates the context record for a new thread, it saves up to the first four thread arguments in the `r0` – `r3` slots, sets the `cpsr` slot to indicate IRQs enabled, sets the `lr` slot to the address of the `userret()` procedure (which properly `kill()`s the thread following the completion of its top-level procedure), and sets the `pc` (program counter) slot to the address of the thread procedure to execute. If more than four arguments need to be passed to the thread procedure (as per the call to `create()`), `setupStack()` places them on the stack following the context record.

When `ctxsw()` is called by `resched()` to reschedule the currently running thread, a context record for the current thread is pushed onto its stack, the stack pointer is loaded with the address of the stack for the next thread, and the context record for the next thread is restored and popped off the stack. More details can be found in the source itself, including comments; `setupStack()` is implemented in `system/arch/arm/setupStack.c` and `ctxsw()` is implemented in `system/arch/arm/ctxsw.S`.

4.5 Pausing, halting, and resetting

While an instance of Embedded Xinu is running, it common to reach a state in which no threads have any work to do. When this occurs, the *null thread* is scheduled to run and must simply wait until an interrupt occurs. This can easily be implemented by an infinite loop that does nothing. However, it is desirable to also put the processor into a low-power state to save energy. For this purpose, ARM processors provide the `wfi` (Wait For Interrupt) instruction, which puts the processor into a low-power state and waits until an interrupt has been received. XinuPi uses this instruction to implement the `pause()` function, which is called by the null thread for this purpose.

A related consideration is how to properly "stop" the operating system when it

Offset	Register	Notes
0x00	r0	First thread argument, caller-save
0x04	r1	Second thread argument, caller-save
0x08	r2	Third thread argument, caller-save
0x0C	r3	Fourth thread argument, caller-save
0x10	r4	Callee-save
0x14	r5	Callee-save
0x18	r6	Callee-save
0x1C	r7	Callee-save
0x20	r8	Callee-save
0x24	r9	Callee-save
0x28	r10	Callee-save
0x2C	r11	Callee-save
0x30	cpsr	Current program status register
0x34	lr	Callee save, link register
0x38	pc	Program counter

Table 3: Format of thread context record which I selected for use in ARM ports of Embedded Xinu. The notes assume the ARM Procedure Call Standard [12]. `r12` and `sp` exist, but they are not included in the context record for reasons noted in the main text.

has finished running, for example when all threads have terminated. Embedded Xinu platforms must implement a `halt()` function for this purpose. Such a function ideally will power off the hardware. However, the Raspberry Pi is designed to be manually unplugged rather than powered off programatically. For this reason, the `halt()` implementation for XinuPi simply disables interrupts and enters an infinite loop, effectively preventing the system from doing anything else.[5]

Still another related consideration is how to programatically reset the system when the **reset** shell command is run. This is not trivial to implement on the Raspberry Pi because the board is intended to be manually power-cycled. However, I found that it is possible to programatically reset the hardware by making use of the *watchdog timer* available on the BCM2835. A watchdog timer is a hardware device including a timer that, once programatically set, must be renewed with a certain period of time to avoid an automatic hardware reset. Watchdog timers are present in many embedded systems to provide a way for the system to automatically return to a working state after a crash. On the Raspberry Pi, an intentional software reset is possible by setting the BCM2835 watchdog timer to a very short delay, then intentionally letting it expire. Unfortunately, the BCM2835 watchdog timer is not documented in Broadcom's datasheet [16]. Therefore, I had to use the Linux driver [43] as a reference for the

[5]Unfortunately, the `wfi` instruction cannot easily be used to place the processor in a low power state indefinitely because, as documented in the *ARM Architecture Reference Manual*[10], any interrupt will wake the processor from `wfi`, even if interrupts are masked (disabled).

software interface to the watchdog timer. The interface is quite simple, but it includes several undocumented "magic numbers", including the register locations, that could not reasonably be determined in any other way.

4.6 UART driver

A *UART* (Universal Asynchronous Receiver/Transmitter) is a hardware device that serves as an intermediary between system-level software and a serial port. Bytes written to the UART by software are transmitted one bit at a time over the serial port, whereas bits received over the serial port are accumulated and provided to software by the UART as bytes. In other words, a UART is a circuit that translates between data in parallel and serial forms. Software operates a UART by reading from and writing to a set of control registers which are typically memory-mapped (e.g. located at a fixed physical memory address). Different UART models have different control registers; common models include NS16550-compatible UARTs and PL011-compatible UARTs.

Serial ports operated via UART are available in many embedded systems in order to provide simple text-based access to the running operating system. Ports of Embedded Xinu to MIPS-based routers such as the Linksys WRT54GL took advantage of two serial ports controlled by NS16550-compatible UARTs. The Raspberry Pi offers a single serial port which software can control using either a partially NS16550-compatible UART or a PL011-compatible UART. Externally, the serial port is accessible through the GPIO header (the 2×13 array of vertical pins clearly visible on the board). The connection can be made through a USB-to-serial converter with 3.3V logic levels. Tx (transmit) is GPIO pin 8 and Rx (receive) is GPIO pin 10. Ground must also be connected to one of the ground pins, for example pin 6. Unlike the Linksys WRT54GL, the Raspberry Pi does not require hardware modification to use the serial port.

Although the Raspberry Pi does include a UART known as the "mini-UART" that is intended to be similar to an NS16550-compatible UART, it is in fact not fully NS16550-compatible [16]. The use of such a nonstandard device in Embedded Xinu would be undesirable when an alternative is available, since standard devices tend to be better documented. Therefore, XinuPi uses the PL011 UART instead.

A team of students at the Rochester of Institute of Technology had conducted preliminary work[32] to port Embedded Xinu to the Raspberry Pi and had already implemented a driver for the PL011 UART, heavily based on the NS16550 UART driver already included in Embedded Xinu. Consequently, Tyler, Farzeen, and I borrowed this code but continued to adapt and improve it.

The PL011 UART available on the Raspberry Pi is documented by Broadcom [16], but there are some errors in this documentation and it is not the primary reference for PL011 UARTs, which is instead a manual published by ARM Ltd. [34]. I have made use of both these resources in order to understand the operation of the PL011 UART and the software interface to it.

The following only briefly summarizes Embedded Xinu's PL011 UART driver, since

a full explanation of the PL011 UART itself would require many pages. The driver logically has two separate interfaces, or modes of operation. The primary interface is an interrupt-driven interface that delegates the communication with the UART itself to the interrupt handler, which is triggered when the UART has received a byte or has become free to transmit another byte. Reads and writes from the software "UART device" then only operate on in-memory buffers that are also accessed by the UART interrupt handler. The other interface presented by the UART driver is a synchronous interface. `kgetc()` polls the UART until a byte has been received, whereas `kputc()` polls the UART until a byte is ready to be transmitted. The synchronous interface is primarily intended for low-level debugging because it does not rely on functional interrupt handling.

More details about the PL011 UART driver can be found in the source itself, located in `device/uart-pl011/`.

Chapter 5

Design & Implementation of USB Support

Like many modern computers, including many embedded systems, the Raspberry Pi relies heavily on USB (Universal Serial Bus) to attach many types of external devices, such as keyboards and mice. In addition, the Ethernet controller on the Raspberry Pi Model B is actually a built-in USB device. As such, USB support is a prerequisite for networking support on the Raspberry Pi.

Embedded Xinu did not previously have USB support. Although several students had worked on it (on other platforms), little progress was made, and it was never a high priority. However, for XinuPi, networking support, and in turn USB support, were high priority features due to the fact that networking is well supported on other Embedded Xinu platforms, such as the Linksys WRT54GL router.

This chapter deals with the design and implementation of the USB subsystem itself and not with drivers for any specific USB devices other than hubs. However, Chapter 6 discusses the design and implementation of a USB device driver for the SMSC LAN9512 Ethernet device, and Chapter 7 discusses the design and implementation of a driver for USB keyboards.

5.1 Difficulties

USB is a plug-and-play bus that was designed for ease of use by end users, inexpensive mass production of compatible hardware, and support for virtually any possible type of device. Unfortunately, simplicity for the developer was not a major goal of USB. Consequently, all versions of the USB specification are lengthy and complicated, and a large amount of code must be written to add USB support to an operating system regardless of the specific USB devices for which support is eventually desired.

The USB 2.0 specification [24] is 650 pages long.[1] Certainly, not all sections are

[1] I do not consider USB 3.0 or later because 2.0 is the revision supported by the Raspberry Pi. Furthermore, due to the backwards compatibility built into USB, devices supporting USB 3.0 can

actually of interest to a USB software implementer, so I have only found it necessary to read a subset of the full specification. But even so, it makes for a time-consuming read. Furthermore, the USB specification does not actually standardize the specific method by which software can send and receive messages over a USB, which is done by communicating with a piece of hardware known as the *USB host controller*. Although this information is of critical importance to software implementers, it is contained in separate specifications for USB host controllers and sometimes is not even documented at all. The latter is the case for the Raspberry Pi, which uses a USB controller[2] known as the "Synopsys DesignWare Hi-Speed USB On-the-Go Controller". The software interface to this USB controller does not conform to any standard, nor is it described in any official (vendor-provided) publicly available documentation. Consequently, implementing a *USB host controller driver* to communicate with this USB controller required examining any and all available sources of information, including the C source code of the vendor-provided Linux driver [49] and basic third-party drivers [20] [39].

Ultimately I was able to implement a working "USB 2.0" subsystem. However, due to the high complexity of USB, not all aspects of USB 2.0 were implemented. Furthermore, despite USB's relatively high level of standardization, it is unreasonable to implement a USB subsystem that is perfectly compatible with all real-world "USB devices". This is because some "USB" devices are not, in fact, fully compliant with the USB standard. This can occur when hardware manufacturers only test their devices for compatibility with a specific USB implementation, such as that in Microsoft Windows, which does things in a very specific way. (Other operating systems such as Linux have had various hacks added for compatibility with such devices.)

5.2 Introduction to USB

Before introducing the USB subsystem I implemented for Embedded Xinu, it is necessary to explain the basics of USB itself. This section explains how a USB-compliant bus, or simply USB, is arranged and how, at a high level, communication with USB devices happens.

5.2.1 Bus topology

Fundamentally, USB is just a way to connect devices to a computer system. A USB accomplishes this by arranging devices in a tree. Each node of the tree is a *USB device*. There are two fundamental types of USB devices: *hubs* and *functions*. USB hubs can have "child" devices in the tree, whereas functions cannot. Hubs can be "children" of other hubs, up to a depth of 7 levels.

typically still be used by USB 2.0 hosts.

[2] I use the phrase "USB controller" rather than "USB host controller" here intentionally, since the Synopsys hardware actually supports both host and device-mode features. However, for my work with Embedded Xinu I was only concerned with host-mode features.

The root node of the tree is the *root hub*, and every USB has one (although it may be faked by the host controller driver, described later).

A USB hub provides a fixed number of attachment points for additional devices. Each such attachment point is called a *port*. A USB port may be internal to the computer system or it may be exposed to the user as a place to plug in a USB cable. From the user's point of view there is certainly a difference between these two manifestations of a USB port, but from the software's point of view there is no difference.

It is also possible that a single physical package, which a naive user might refer to as a "USB device", actually contains an integrated USB hub to which one or more USB devices are attached. Such physical packages are called *compound devices*. A common example of a compound device is one of Apple's USB keyboards that provides a USB port to attach a mouse.

Since USB is a dynamic bus, USB devices can be attached or detached from the USB at any arbitrary time. Detaching a hub implies detaching all child devices.

5.2.2 Devices

Due to the generality of USB, a USB device can be virtually anything at all. This is made possible in part by a highly nested design:

- A device has one or more *configurations*.
- A configuration has one or more *interfaces*.
- An interface has one or more *alternate settings*.
- An alternate setting has one or more *endpoints*.

Every device, configuration, interface, and endpoint has a corresponding *descriptor* that can be read by the USB software to retrieve information about the described entity in a standard format.

Although this highly nested design allows for complex devices, most devices are relatively simple and have just one configuration. Furthermore, common devices have only one interface.

5.2.3 Host controllers

USB is a polled bus, so all transfers over the USB must be initiated by the *host*. The term "host" in this context means the USB software as well as the USB host controller, the latter of which is the hardware responsible for actually sending and receiving data over the USB and maintaining the root hub.

Since the USB specification in many cases does not standardize the exact division of tasks between software and hardware, it is often unclear whether hardware or software is responsible where the specification says "host". The essential thing to

know is that the place where the USB software meets the USB hardware is in the USB host controller driver, which operates the USB host controller. Some USB host controllers present standard interfaces (such as UHCI, OHCI, or EHCI— all defined in specifications separate from the USB specification itself) to software. Others do not present a standard interface, but instead have vendor-provided documentation and/or a vendor-provided driver. Obviously, a standard interface is highly preferred when independently implementing a USB host controller driver.

5.2.4 Transfers

To communicate with devices, a USB host must send and receive data over the USB using *transfers*. A transfer occurs to or from a particular endpoint on a particular interface of a particular device. Each endpoint can be used for exactly one of four predefined types of transfers:

- *Control transfers* are typically used for device configuration. There are two main unique features of these transfers. First, a special packet called SETUP is always sent over the USB before the actual data of the control transfer, and software needs to specify the contents of this packet. Second, every device has an endpoint over which control transfers in either direction can be made, and this endpoint is never explicitly listed in an interface descriptor.
- *Interrupt transfers* are designed for time-bounded transmission of small quantities of data (e.g. data from a keyboard or mouse).
- *Bulk transfers* are designed for reliable (with error detection) transmission of large quantities of data with no particular time guarantees (e.g. reading and writing data on mass storage devices).
- *Isochronous transfers* are designed for periodic transmission of data with no error detection (e.g. video capture).

Each USB transfer has a direction that is either device-to-host (IN) or host-to-device (OUT). An endpoint may be restricted to transfers in a specific direction, or it may be bidirectional.

5.2.5 Speeds

USB supports multiple transfer speeds:

- 1.5 Mbit/s (Low Speed) (USB 1.0+)
- 12 Mbit/s (Full Speed) (USB 1.0+)
- 480 Mbit/s (High Speed) (USB 2.0+)

- 5000 Mbit/s (Super Speed) (USB 3.0+)

Due to the need to maintain backwards compatibility with legacy devices, the USB host controller driver sometimes must take transfer speeds into account. At minimum, it must be aware that transfers to or from devices attached at Low or Full Speed must be performed as a series of *split transactions* between the host and a High Speed hub intermediary. This design allows Low or Full Speed transfers to occur without significantly slowing down the portion of the bus operating at High Speed.

5.3 Embedded Xinu's USB subsystem

Now that I have presented some general information about USB, it should be possible to understand the USB subsystem I implemented for Embedded Xinu.

5.3.1 Overview

Embedded Xinu's USB subsystem is divided into three components, or drivers:

- 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.

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

In this software architecture, all USB device drivers, including the USB hub driver, interact with USB devices through the USB core driver, which in turn provides an abstraction layer around the USB host controller driver. This separation of concerns is fairly standard across all USB-enabled operating systems, since it mirrors how USB was designed.

Embedded Xinu's USB subsystem understands USB's dynamic tree topology and does not assume a fixed set of devices. Furthermore, it permits the use of control, interrupt, and bulk transfers through a specialized API. Inevitably, these features contrast with the main Embedded Xinu device model, which, as described in Section 2.1.3, assumes a static set of devices on which I/O may be performed through `read()` and `write()` operations. However, Chapter 6 and Chapter 7 describe USB device drivers that export useful functionality of their underlying USB devices as standard Embedded Xinu devices.

All parts of the USB subsystem were implemented entirely by me. However, I borrowed design ideas from the USB specification [24], prior knowledge about how other operating systems such as Linux implement USB support, and "Chadderz Simple USB Driver for Raspberry Pi" [20]. For the host controller driver in particular, I describe additional sources of information in Section 5.3.4.

Due to its high complexity (upwards of several thousand lines of code), I cannot mention all details of the USB subsystem in this thesis. The reader can find additional details in the source code itself as well as in the corresponding documentation that I wrote for the Embedded Xinu project itself. The latter documentation is partly redundant with this thesis, but I wrote it for a different audience and it is sometimes concerned with different aspects of USB support.

5.3.2 USB core driver

Embedded Xinu's USB core driver is responsible for maintaining the USB device model (including the tree structure) and providing a framework in which USB device drivers can be written. The USB core driver does not itself communicate with the USB hardware but instead relies on a platform-dependent USB host controller driver for this purpose. The USB core driver can alternatively be viewed as the software responsible for all USB-related tasks that depend on neither specific USB devices nor specific USB host controller implementations.

When a new USB device is detected by the USB hub driver, the USB hub driver calls `usb_attach_device()` in the USB core driver to perform tasks common to every USB device, such as setting the configuration and device address and reading the device descriptor. The USB core driver then attempts to "bind" the new USB device to each registered USB device driver, in turn. Drivers not supporting the device are expected to return a certain status code so that the next driver can be queried.

The USB core driver provides the functions `usb_submit_xfer_request()` and `usb_control_msg()` to allow USB device drivers to communicate with the USB devices they are controlling. `usb_submit_xfer_request()` asynchronously performs the specified type of transfer over the USB to or from the specified endpoint on the specified USB device, whereas `usb_control_msg()` wraps around `usb_submit_xfer_request()` to perform a synchronous control transfer to or from the specified endpoint on the specified USB device.

The initialization function for the entire USB subsystem, `usbinit()`, is located in the USB core driver. It is called during operating system initialization and is responsible for registering the USB hub driver, initializing the host controller driver, and initializing the USB device model by setting up a USB device structure for the root hub and binding it to the USB hub driver. `usbinit()` is not itself responsible for enumerating the entire USB — that is, discovering all devices attached to it. That instead proceeds asynchronously with the help of the hub driver.

Despite being usable for supporting various USB devices, Embedded Xinu's USB core driver still has many limitations compared to the equivalent in other operating

systems such as Linux. For example, it does not support isochronous transfers, per-interface USB drivers, multiple concurrent USBs, intelligent power management, or any features introduced later than USB revision 2.0. However, none of these features currently have a clear need, and since Embedded Xinu is an instructional operating system it is desirable to omit them for now.

The code of the USB core driver is mostly located in `device/usb/usbcore.c`. Optional (but useful) functionality dealing with human-readable messages and strings has been factored out into `device/usb/usbdebug.c`.

5.3.3 USB hub driver

Embedded Xinu's USB hub driver is an example of a USB device driver, but it is somewhat special in that it controls USB hubs, which have a unique role in USB. In the USB topology (see Section 5.2.1), the path from the host to any function passes through a nonempty sequence of hubs, each of which must rely on the hub driver to enumerate the devices attached to it. Therefore, support for any USB device whatsoever relies on a hub driver being available. As such, the hub driver is very much part of the base USB subsystem itself, and I placed its source code in the same directory as the USB core driver to prevent confusion about whether it is an optional component or not. The source file is `device/usb/usbhub.c`.

The initial entry point of the USB hub driver is `hub_bind_device()`, which is called by the USB core driver when it has configured a newly attached USB device that may be a hub. `hub_bind_device()` is responsible for checking if the device is a hub, and if so, then doing hub-specific setup, including one-time driver initialization, reading the hub descriptor, powering on the ports, and submitting an asynchronous USB interrupt transfer request to the hub's status change endpoint.

Everything else the hub driver does happens asynchronously as a response to a status change request being completed. Every USB hub has exactly one interrupt IN endpoint called the *status change endpoint*. The hub responds on this endpoint whenever the status of the hub or one of the hub's ports has changed— for example, when a USB device has been connected or disconnected from a port.

At the hardware level, when a hub has data to send on its status change endpoint, an interrupt will come in from the USB host controller. This eventually will result in the status change transfer being completed and `hub_status_changed()` being called. Thus, the detection of status changes is interrupt-driven and is not implemented by polling at the software level, at least in the hub driver. (At the hardware level, USB is still a polled bus, but the host controller hardware and/or driver handles that.) Upon detecting a status change on one or more ports on a hub, the hub driver then must submit one or more control messages to the hub to determine exactly what changed on the affected ports. However, the hub driver defers this work by passing it to a separate thread in order to avoid doing too much synchronous work in interrupt handlers. When this separate thread finally determines the statuses of the hub's ports, it calls into the USB core driver to react to the attachment or detachment of any USB devices.

5.3.4 USB host controller driver

A USB host controller driver is responsible for actually sending and receiving data over the USB by making use of the platform-dependent host controller hardware. The purpose of this type of driver is to isolate differences in USB host controllers from all other code dealing with USB. This is an appropriate software architecture since it makes it easier to support USB on multiple platforms. However, as of this writing, the only USB host controller driver implemented in Embedded Xinu is the one I wrote to control the Synopsys DesignWare Hi-Speed USB On-the-Go Controller used on the Raspberry Pi.

In the software architecture I designed for Embedded Xinu's USB subsystem, USB host controller drivers must implement three functions. The `hcd_start()` and `hcd_stop()` functions must initialize and de-initialize the USB host controller, respectively. The `hcd_submit_xfer_request()` function must submit a USB transfer, represented by a `struct usb_xfer_request`, to be asynchronously completed with the help of the host controller. A `struct usb_xfer_request` contains the device address, endpoint number, data buffer, number of bytes to transfer, optional SETUP data, and a pointer to a callback function. The host controller driver executes the callback function when the transfer has completed or failed. Additional documentation for the USB host controller driver interface can be found in the header declaring it (`include/usb_hcdi.h`).

The remainder of this section deals specifically with the host controller driver for the Synopsys USB controller used in the Raspberry Pi. Broadcom, the manufacturer of the BCM2835 System-on-a-Chip, briefly describes the Synopsys USB controller and notes the location of its register interface in the memory space of the BCM2835 [16]. However, Broadcom provides no documentation about the actual registers of the Synopsys USB controller and instead redirects the reader to a document provided by Synopsys Inc. Unfortunately, the Synopsys document is only available to Synopsys customers, which include Broadcom but not "end users" such as myself. Because of this as well as the fact that the interface to the Synopsys USB controller is not otherwise documented in a standard such as EHCI, the Synopsys USB controller is de facto undocumented.

Writing a driver for undocumented hardware is inevitably very difficult, and to do so the programmer must rely on any and all sources of information that may be available. In the worst case, this includes only the black-box behavior of the hardware itself and the machine code of closed source drivers for other operating systems. But fortunately, Synopsys has released an open source driver [49] for their USB controller. It supports Linux, and consequently it is the driver used when typical Raspberry Pi users run Linux-based operating systems on the board. This driver has not, however, been accepted into the mainline Linux kernel — that is, the "official" Linux kernel maintained by Linus Torvalds. Instead, it is only included in the fork of Linux being maintained for the Raspberry Pi until all Raspberry Pi support has been "mainlined".

In theory, a working open-source driver is an ideal replacement for real documenta-

tion because the source code is unambiguous, unlike documentation written in a natural language such as English, and necessarily operates the device successfully in some way. However, in practice, reading and understanding the Synopsys driver was very difficult due to its high level of complexity. Despite its conceptually simple task of relaying USB messages between software and hardware, it still contains approximately 37,000 lines of C source code. This 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.

Over the course of several weeks and largely concurrent with my work on the USB core driver (Section 5.3.2) and USB hub driver (Section 5.3.3), I worked to extract the basic hardware-software interface information from the Synopsys driver necessary to create an independent implementation. I also referred to a limited number of drivers, with varying levels of functionality, that have recently been implemented for the same hardware [20][39]. However, these third-party drivers all seemed to be based on the Synopsys driver.

One of the most useful files in the Synopsys driver was `dwc_otg_regs.h`, which declared the register interface provided by the USB controller, including the contents of each register. However, detailed information about how the registers actually must be used usually was not present in the comments, so for that I had to find and examine the actual code which accessed them.

Ultimately, my contribution is a working, well-commented driver[3] to control the USB controller on the Raspberry Pi that runs in the lightweight Embedded Xinu environment and is approximately 20 times shorter than the corresponding Linux driver. The difference in code length arises because my driver only supports only a carefully-chosen subset of features and does away with unnecessary abstraction layers that were present in the Synopsys driver. My driver also does not contain logic from the Linux driver that I was unable to understand under the time constraints and/or found was unnecessary to operate the device based on empirical tests. (The extent to which this matters is an open question. Presumably, most of the unimplemented logic is not truly necessary in a basic implementation, since my implementation is already functional enough to support several device drivers.)

The entry point of my driver for the Synopsys USB controller is `hcd_start()`, which must perform the following tasks to prepare the controller to be ready to use:

1. Enable power to the USB controller by telling the VideoCore to do so, via the same memory-mapped "mailbox" message-passing mechanism the framebuffer (graphics) driver uses (see Section 9.2).

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

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

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]

4. Software not performing USB transfers completely synchronously must enable interrupts from the USB controller, both in the AHB Configuration Register in the USB controller and in the BCM2835 ARM Interrupt Controller described in Section 4.2.2. At minimum, software must enable Host Channel interrupts, which are required for the software to be notified when USB transfers have completed, and Host Port interrupts, which are required for the software to emulate the root hub.

After the Synopsys USB controller is ready to use as a result of successful execution of `hcd_start()`, the `hcd_submit_xfer_request()` function can be called to start a transfer over the USB to or from a specific endpoint on a specific USB device. To actually perform such a transfer, software must, briefly, do the following:

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

2. Wait for an interrupt to occur from the USB Controller. (As noted in Section 4.2.2, this uses BCM2835 shared IRQ line 9.)

3. Check the 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 Synopsys USB controller 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.

[4] I spent a long time debugging this problem. It is unclear to me why the parameters are wrong by default; perhaps Broadcom made a mistake when they built (selected parameters for) the Synopsys hardware.

- Although USB is a polled bus, as far as I can tell the Synopsys USB controller 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 controller halts the channel and sets the "NAK response received" flag, thereby forcing software to explicitly delay for an appropriate interval for polling.

- The Synopsys USB controller seems to have some special scheduling requirements related to periodic transfers and frame boundaries which I do not yet fully understand. The Embedded Xinu driver currently works around certain problems by automatically retrying certain transfers if they fail.

To provide more technical information about the Synopsys USB controller, I have written detailed documentation for the essential registers in Embedded Xinu's header file declaring them (`system/platforms/arm-rpi/usb_dwc_regs.h`). However, it was only possible to document the functionality actually used by the Embedded Xinu driver, so this documentation cannot be used to, for example, write a drop-in replacement for Synopsys' Linux driver that will have no feature regressions.

5.3.5 Development and testing

During early development of the USB subsystem, I first implemented support for synchronous control transfers. With only this in place, I built early versions of the core, hub, and host controller drivers and was able to enumerate the USB on the Raspberry Pi Model B (which as noted in Section 3.2.4 contains an internal hub) and perform generic USB-defined configuration of attached devices.

Slightly later, I added support for interrupt transfers and made all transfers asynchronous (interrupt-driven). This was accompanied by updates to all three drivers. At this point, due to the design of USB (especially its use of hubs), I was able to test both control and interrupt transfers at multiple speeds, as well as the hub driver, simply by plugging in different types of USB devices, such as mice, keyboards, and flash drives. This was true even though no actual device drivers had been implemented yet. The USB subsystem also received more testing when I tested the SMSC9512 (Chapter 6) and USB keyboard (Chapter 7) drivers. For example, the SMSC9512 driver was the first code that used bulk transfers and transfered large amounts of data over the USB.

In any case, I did not have the time or resources to do formal compliance testing of the USB subsystem. Furthermore, since the USB subsystem is strongly coupled to the hardware, I was unable to implement any automated tests. However, automated tests of higher-level functionality, such as networking using the SMSC9512 device, rely on the underlying USB subsystem functioning correctly.

Chapter 6

Design & Implementation of SMSC9512 Driver

6.1 Background and device information

As mentioned earlier, the Raspberry Pi Model B has one Ethernet port, supported through an integrated *LAN9512 USB 2.0 Hub and 10/100 Ethernet Controller* designed by SMSC[1], located on the USB. This device consists of a USB hub with three ports, two of which correspond to the externally visible USB ports on the Raspberry Pi and one of which is permanently connected to a vendor-specific class device that implements the Ethernet controller. No WiFi or other network interfaces are available. Therefore, USB support, including a hub driver, was a prerequisite to networking support on the Raspberry Pi in order to make XinuPi competitive with previous Embedded Xinu platforms, such as the WRT54GL router. However, once USB support was implemented, as detailed in Chapter 5, it became possible to implement the driver for the SMSC Ethernet controller.

The hub portion of the SMSC9512 is designed to be controlled by a USB hub driver, and Embedded Xinu's hub driver (described in Section 5.3.3) serves this purpose. Therefore, the rest of this chapter deals specifically with the vendor-specific class device attached to this hub that implements the Ethernet controller itself, which shall henceforth be referred to simply as the SMSC9512.

The role of the SMSC9512 is to provide a bridge between the USB and the Ethernet. This is the same role as commercially available "USB Ethernet adapters", which are seen by software in almost exactly the same way and some of which may even use the same chip. The only the difference is that on the Raspberry Pi, the SMSC9512 is attached to the USB internally.[2]

[1] SMSC stands for Standard Microsystems Corporation, now acquired by Microchip Technology.

[2] The SMSC9512 chip is actually clearly visible on the Raspberry Pi Model B as the largest chip other than the BCM2835. It may be theoretically possible to remove or replace it with specialized tools, but it clearly is intended to be part of the board itself.

Ethernet controllers operate by sending and receiving *Ethernet frames*, which are the physical-layer wrappers around networking packets (which themselves may implement other protocols, such as the Internet Protocol). The operation of a USB Ethernet adapter such as the SMSC9512 is therefore primarily concerned with receiving Ethernet frames by transferring them over the USB from the USB Ethernet adapter to the USB host and sending Ethernet frames by transferring them over the USB from the USB host to the USB Ethernet adapter. Typically, these transfers occur to or from bulk endpoints on the USB device, and the SMSC9512 is no exception to this.

However, actually writing a driver to control such a device requires more specific details of how to configure the device and what protocol is used to send and receive packets. Although there does exist a USB Communications Device Class which standardizes the operation of USB Ethernet adapters, not all real devices exactly implement this standard, and the SMSC9512 is one of them.

SMSC provides a datasheet for the SMSC9512 [46]. However, it primarily covers hardware details, such as the pin layout of the chip, and it only very briefly mentions the software interface to the device. Attempts to contact SMSC9512 for access to internal documentation were unsuccessful. Consequently, there is no publicly available documentation for the software interface to the SMSC9512, placing it in a similar predicament as the Synopsys USB controller described in Section 5.3.4.

Fortunately, there is a Linux driver [33], originally written by SMSC themselves, for the SMSC9512 and other devices in the SMSC95xx series.[3] Linux also contains a driver that implements generic functionality required by all USB network devices [31]. Each of these drivers contains about half the code needed to operate the SMSC9512, excluding the much larger amount of code in the USB subsystem itself. Therefore, in order to implement my own driver for the SMSC9512, I had to examine these two Linux drivers to determine the software interface to the device. This was not trivial because the drivers contained many details that ultimately turned out to be irrelevant. In addition, many registers and flags the SMSC Linux driver used to configure or operate the device were not explained, so I had to test some empirically to see what they really did. The following section describes the driver I implemented and how it controls the SMSC9512 to offer networking support for XinuPi.

6.2 Embedded Xinu driver

The source code of the SMSC9512 driver I implemented is located in `device/smsc9512/`. Like the Ethernet drivers for other Embedded Xinu platforms, it provides a static Ethernet device table entry `ETH0` which the networking module can `read()` packets from and `write()` packets to. These operations get routed to `etherRead()` and `etherWrite()`, respectively. However, as a USB device driver, the SMSC9512

[3]The FreeBSD operating system also contains a driver for the SMSC9512, but based on the source code itself it was written by a third-party programmer based on the Linux driver and did not appear to be much more useful than the Linux original.

driver has many differences from the other Ethernet drivers that have been implemented in Embedded Xinu.

6.2.1 Initialization

The operating system initializes SMSC9512 driver by calling `etherInit()`, which is configured in `xinu.conf` as the `init()` function for `ETH0`. However, because `ETH0` is a static device entry in contrast to the underlying USB device (which is dynamic, even though the device is non-removable on the Raspberry Pi, as there is no guarantee as to when it is actually detected), `etherInit()` can actually do little except register the SMSC9512 driver with the USB subsystem so that later initializations can be performed when the device is actually detected.

Some of these later initializations occur in `smsc9512_bind_device()`, which checks if a newly attached USB device is the SMSC9512, and if so, does some preliminary configuration (setting the MAC address to a value that is either randomly generated or based on the board's serial number, and enabling multiple packets per USB transfer) and links the `struct ether`, which represents the SMSC9512 at the networking level, with the corresponding `struct usb_device`, which represents the SMSC9512 at the USB level. The driver configures the SMSC9512 by writing to registers in its register set, which are accessible through vendor-specific control transfers performed to or from the SMSC9512's default control endpoint. Doing this requires knowing the locations and meanings of the registers. In the file `device/smsc9512/smsc9512.h` I have attempted to document some of the registers I used in the Embedded Xinu driver, since the corresponding definitions in the Linux driver contain no documentation.

Still more initializations of the SMSC9512 occur when the operating system actually opens the Ethernet device, which results in a call to `etherOpen()`. This function waits for the SMSC9512 to be asynchronously attached (if it wasn't already), sets aside buffer space for sent and received packets, prepares one `struct usb_xfer_request` each for bulk IN and bulk OUT USB transfers, submits the bulk IN transfer for receiving packets, and finally enables Rx (Receive) and Tx (Transmit) on the device.

6.2.2 Sending and receiving packets

At any point after the initializations described above have been completed, the bulk IN transfer from the SMSC9512 may complete to indicate that one or more Ethernet frames were received. (At a lower level, this occurs when the USB host controller interrupts the CPU to indicate successful completion of the transfer; however, this detail can be considered specific to the USB subsystem.) When this occurs, the USB subsystem executes the callback `smsc9512_rx_complete()`, which extracts the Ethernet frames from the USB data and buffers them from for later retrieval by `etherRead()`. The

```
$ ping 192.168.1.1 -c 10 -s 500 -i 0
PING 192.168.1.1 (192.168.1.1) 500(528) bytes of data.
508 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.771 ms
508 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.689 ms
508 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.696 ms
508 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.707 ms
508 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=0.705 ms
508 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=0.719 ms
508 bytes from 192.168.1.1: icmp_seq=7 ttl=64 time=0.711 ms
508 bytes from 192.168.1.1: icmp_seq=8 ttl=64 time=0.712 ms
508 bytes from 192.168.1.1: icmp_seq=9 ttl=64 time=0.711 ms
508 bytes from 192.168.1.1: icmp_seq=10 ttl=64 time=0.699 ms

--- 192.168.1.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 7ms
rtt min/avg/max/mdev = 0.689/0.712/0.771/0.021 ms, ipg/ewma 0.813/0.726 ms
```

Figure 9: Example of running a manual networking test to verify correct operation of the underlying SMSC9512 driver. The **ping** command, which is shown on the first line and was run on a separate computer running Linux, sent a number of ICMP Echo Request packets to the Raspberry Pi running Embedded Xinu. The **-i 0** option instructed **ping** to send the packets with no delay (i.e. as fast as possible), in order to increase the chance of detecting networking problems. As shown by the statistics summary, all packets were successfully returned (as ICMP Echo Replies) very quickly. This demonstrated Embedded Xinu's SMSC9512 driver both receiving and sending packets. Note: actual tests were done with higher packet counts to further increase the chance of detecting any problems.

format in which the Ethernet frames are embedded in the USB data is device-specific and had to be determined by examining the Linux implementation.

When the network subsystem calls `etherWrite()` to request that an Ethernet frame be sent out on the network, the SMSC9512 driver sends the data to the SMSC9512 using a bulk OUT USB transfer. Like incoming packets, outgoing packets must be embedded in the USB data in a device-specific format that had to be determined by examining the Linux implementation.

6.2.3 Testing

I primarily tested the SMSC9512 driver manually by performing stress-tests with sending and receiving packets. I connected the Raspberry Pi (running Embedded Xinu) to another computer (running Linux) using an Ethernet cable, then configured static IP addresses and routing information on both ends. I then used the **ping** command on Linux to send repeated ICMP Echo Request packets to the Raspberry Pi using various packet sizes and repeat rates. The expected behavior was that Embedded Xinu promptly respond to each packet with an ICMP Echo Reply packet. **ping** reported statistics

dealing with packet loss, packet corruption, and latency, and I checked these statistics for possible issues. In the latest version of the driver, there are no known issues.[4] Figure 9 shows an example of one such test. In addition, the SMSC9512 driver received more testing when I tested additional networking features, such as the DHCP (Section 8.3) and TFTP (Section 8.4) clients.

I also tested the SMSC9512 driver automatically using an Ethernet driver test program that is included in Embedded Xinu's test suite. This test relies on placing the Ethernet device in loopback mode, which is a hardware mode that causes transmitted packets to "loop back" to the receiver. Specifically, the test program uses this mode to send packets to itself through the network adapter without relying on an external computer. Although the Linux driver[33] never places the SMSC9512 in loopback mode, it defines, with no explanation, a value `MAC_CR_LOOPBK_`. I found, empirically, that the device enters loopback mode when software writes this value to the MAC Control Register. This made it possible to run the automated test, which succeeds in the latest version of the driver.

6.2.4 Limitations

Since Embedded Xinu is an instructional operating system, and for the sake of time, I did not implement software support for all features supported by the SMSC9512 hardware. Some extra features which the Linux driver supports but my Embedded Xinu driver does not include:

- Support for other potentially compatible SMSC LAN95xx devices
- Dynamically detaching the device[5]
- Reading from attached EEPROM[5]
- VLANs
- TCP/IP checksum offload
- Various PHY (physical layer) features, such as support for Wake-on-LAN and suspend

[4] This section is not meant to convey the entire development process of the SMSC9512 driver and the underlying USB subsystem. In particular, I spent a long time fixing a bug in the underlying USB subsystem that showed up when testing the SMSC9512 driver.

[5] Not possible on the Raspberry Pi.

Chapter 7

Design & Implementation of USB Keyboard Driver

7.1 Background

With a USB subsystem for Embedded Xinu in place (see Chapter 5), it became possible to implement support for useful USB devices. One very common and useful type of USB device is a USB keyboard. Although Embedded Xinu primarily relies on simple serial ports (see Section 4.6) for human interaction, the USB support available on the Raspberry Pi as well as the graphics support (see Section 9.2) made possible a keyboard-and-monitor setup as an alternate way for humans to interact with the Embedded Xinu operating system. This section describes the implementation of a driver for USB keyboards for the input half of this setup.

Initially, Tyler Much was responsible for writing the USB keyboard driver, but little progress was made for various reasons, such as the unexpectedly high difficulty of the task. As we discovered, USB keyboards are normally supported through the USB HID (Human Interface Device) specification [59], which provides a USB device class intended to represent arbitrary input devices including not only keyboards, but also mice, trackballs, joysticks, knobs, VCR remote controls, data gloves, steering wheels, and thermometers. The above device types do not have specific input data protocols defined, but rather each HID device may speak its own protocol, as specified in a declarative format in a structure called the *device descriptor* that the HID device must provide when queried. Consequently, HID devices are intended to all be supported by a single HID driver in combination with a sophisticated input subsystem that can handle all these different types of devices. The HID driver must be able to parse and understand arbitrary HID descriptors which necessarily requires many thousands of lines of code.[1]

[1]The Linux HID driver (which is actually split into multiple parts) contains upwards of 4000 lines of code (not even counting headers and device-specific quirks), and the Linux input subsystem contains thousands more.

Tyler had written some initial code to parse HID descriptors (with the help of some code I had provided for actually communicating with the USB keyboard using the USB subsystem), but when I returned to the code later to actually get the driver working, I decided to take a simpler approach. Since the designers of the USB HID specification knew that it would be very complicated to get a simple keyboard working, they actually had defined an alternative protocol to receive data from USB keyboards. This alternate protocol is defined in the USB HID specification itself and is referred to as the *boot protocol*. As documented in Section 7.2 of the USB HID specification [59], software can enable the boot protocol on a USB keyboard by sending a specific request to the keyboard's default control endpoint. Although it is seemingly not required that all "USB keyboards" support the boot protocol, it seems to be widely implemented due to its use in the BIOSes of desktop and laptop PCs.

Following enabling of the boot protocol, software can read the report, or packet of input data, from the keyboard by performing an interrupt transfer from its interrupt IN endpoint. The hardware completes this transfer when a key on the keyboard has been pressed or released, or after a certain amount of time has elapsed. The resulting report data contains information about keys currently pressed. Specifically, modifier keys such as Shift and Control are represented as a bitmask, and other keys are represented as HID Usage IDs which can interpreted with the help of the USB HID Usage Tables specification [60].

7.2 Driver

The working Embedded Xinu USB keyboard driver is located in `device/usbkbd/`. Like other Embedded Xinu device drivers such as the various UART drivers, the USB keyboard driver provides one or more device table entries, such as `USBKBD0`, `USBKBD1`, etc. Applications can read characters from these devices using the regular `getc()` or `read()` system calls. Internally, the implementations of these functions make use of a buffer that is written to by `usbKbdInterrupt()` when key-presses are detected. Since the USB keyboard driver provides data as 8-bit characters, it has limited support for special keys that do not correspond to printing characters (e.g. F1, Page Up, ...). Currently, the driver simply ignores most of these special keys. Note that these limitations are due to the Embedded Xinu device model rather than the keyboard boot protocol or even the USB keyboard driver per se — that is, the driver *does* receive information about presses of keys like "Page Up", but it has no way to report it to applications.

Since USB is a dynamic bus, there is no guarantee that all supported USB keyboard devices are attached at all times. I handle this in the driver by dynamically binding and unbinding USB devices from static `USBKBD` devices. The first USB keyboard plugged in becomes `USBKBD0`, the second (while the first is still plugged in) becomes `USBKBD1`, etc. Furthermore, attempts to read from a USB keyboard device that does not currently have a physical keyboard bound to it simply block rather than returning

an error code. This means that an application that attempts to get input from a USB keyboard that is not actually plugged in will simply be put to sleep (by waiting on a semaphore) until a keyboard is, in fact, plugged in.

The `sysinit()` function, which contains initialization logic common to all Embedded Xinu platforms, calls `init()` on each device table entry. This results in a call to `usbKbdInit()` for each USB keyboard device entry. On the first such call, the USB keyboard driver registers itself with the USB subsystem as a USB device driver. `usbKbdInit()` does *not* wait for a keyboard to actually be attached before returning.

When the user attaches a USB device or when an internal USB device is first detected, the USB subsystem calls `usbKbdBindDevice()`, which is one of the callbacks that the USB keyboard driver provided when it registered itself with the USB subsystem. If the new USB device has a HID interface supporting the keyboard boot protocol (i.e. "is a USB keyboard"), the USB keyboard driver starts a recurring USB transfer from the keyboard's interrupt IN endpoint. When each such transfer completes (due to a key being pressed or released, or a timeout elapsing), the USB subsystem executes the driver-provided callback `usbKbdInterrupt()`, which parses the transfer data and buffers any new characters for later retrieval by reads from the USB keyboard device.

Since the USB keyboard driver depends on a physical keyboard, it was not practical to implement automated tests for it. Therefore, I primarily tested it by configuring a shell that reads its input from `USBKBD0`, then plugging in a keyboard and typing on it. I verified I had entered the keymap correctly by typing every possible key, both with and without Shift pressed.[2] Slightly later, I modified Embedded Xinu's default `main()` function to start up a shell using a USB keyboard and monitor when both devices are included in the build. This means that a user can, by default, interact with XinuPi using either a keyboard-and-monitor setup or a bidirectional serial port.

[2]This simple test is not meant to convey the entire development process. For example, even before I started writing the USB keyboard driver, I tested communicating with several USB keyboards using standard USB messages in order to verify that the USB subsystem itself could communicate with Low Speed devices.

Chapter 8

Design & Implementation of Network Bootloader

8.1 Motivation

Previous ports of Embedded Xinu have been used in laboratory environments where students can run experimental kernels on real back-end hardware [17] [18]. This functionality relied on the ability of the firmware to perform *network booting* — that is, loading the operating system kernel from over the network. This feature was supported in the CFE (Common Firmware Environment) present on the Asus and Linksys wireless routers to which Embedded Xinu had been ported. In addition, the CFE provided an interactive shell through which commands could be issued, for example to load and execute a kernel from over the network.

However, network booting support is not known to be available in the firmware of the Raspberry Pi. The Raspberry Pi firmware, which is provided by Broadcom and executes on the VideoCore co-processor (see Section 3.2.2), automatically boots the ARM processor using a file `kernel.img` located in a FAT (File Allocation Table) filesystem on the memory card. There is no known way to specify an alternate boot location, and the firmware is not known to even have the USB support that would be a prerequisite to operating the SMSC LAN9512 network device on the Raspberry Pi.

But regardless of the firmware support, it is still possible to write software that runs on the ARM processor and boots the Raspberry Pi over the network. This chapter describes how I implemented this support in XinuPi so that XinuPi can serve as a bootloader for itself. Note that this network booting support depends on the networking support I implemented (see Chapter 6) and therefore provides another motivation for that aspect of porting Embedded Xinu to the Raspberry Pi.

8.2 Overview

At a high level, network booting can be divided into three main steps. The first step is to configure a network interface with Internet-layer protocol information. This typically means assigning an IPv4 (Internet Protocol version 4) address, netmask, and gateway. These values can be assigned either statically or dynamically. If dynamic network configuration is desired, then the standard DHCP (Dynamic Host Configuration Protocol) can be used. In common implementations of network booting, including those used on most routers and personal computers, DHCP is the chosen method because it avoids making assumptions about the specific network configuration and also includes a way to pass information to the client about which server and filename to use for the second step (downloading the new kernel).

The second step in network booting is to download a new kernel from a remote host over the network. This theoretically can be accomplished using one of a number of file-transfer protocols such as HTTP. Although supported by Embedded Xinu, protocols such as HTTP that run over UDP and TCP are too complex to have been widely implemented in firmware. Instead, a simplified file transfer protocol called TFTP (Trivial File Transfer Protocol) is usually used instead.

The third and final step in network booting is to irreversibly pass control from the currently executing kernel (the bootloader) to the new kernel. The possible ways of doing this depend on the specific hardware.

In order to achieve high levels of flexibility, standardization, and protocol simplicity, I implemented the network bootloader using the DHCP and TFTP protocols. Neither of these protocols was supported in the development version of Embedded Xinu. Although both DHCP and TFTP have been supported by past versions, I ended up having to rewrite the code to ensure high-quality implementations with the needed features. In addition, I implemented a procedure called `kexec()` that transfers control to a new kernel on the Raspberry Pi. (*kexec* is short for *kernel execute* and is the term used by the Linux kernel for its equivalent feature.)

8.3 DHCP client

DHCP ("Dynamic Host Configuration Protocol") is a protocol for IPv4 autoconfiguration that is widely used on the Internet and in local area networks. DHCP is an open standard specified in RFC 2131 [26].

DHCP is a client-server protocol concerned with two types of actors:

- *DHCP clients*, which are hosts attempting to obtain an IPv4 networking configuration.
- *DHCP servers*, which keep track of the state of one or more subnets and assign IPv4 networking configurations to clients.

A typical DHCP session proceeds as follows:

1. A DHCP client broadcasts a `DHCPDISCOVER` message on the network to announce its interest in joining an IPv4 network. The `DHCPDISCOVER` is sent to the hardware broadcast address because no Internet-layer information is available.

2. The DHCP server responds with a `DHCPOFFER` message to offer an IPv4 networking configuration to the DHCP client. The offered IPv4 networking configuration must include an IPv4 address that is not currently known to be in use on the network.

3. The DHCP client sends a `DHCPREQUEST` message to accept an offer from a DHCP server.

4. The DHCP server responds with `DHCPACK` to confirm to the client that it has committed the reservation of the IPv4 networking configuration being assigned to the client.

The above is a simplified description and does not cover certain aspects, such as DHCP forwarding and rebinding DHCP leases. A full state diagram for DHCP as well as the formats of the protocol messages can be found in RFC 2131 [26]. Since DHCP is a widely deployed, well-documented protocol, there is little reason to document it in full in this thesis.

Although a DHCP client was available for a prior version of Embedded Xinu, it contained various problems and flaws, including:

- The old DHCP client was not source-compatible with the current version of Embedded Xinu, primarily due to changes in the network subsystem that occurred before I started working with the code.

- The old DHCP client did not re-send `DHCPDISCOVER` packets if no `DHCPOFFER` was received in a certain amount of time. In other words, it was not sufficiently tolerant of networking problems such as packet loss.

- Although the old DHCP client provided access to the IP address, netmask, and gateway assigned by the DHCP server, it did not provide access to the bootfile name and TFTP server IP address. The latter two pieces of information are used specifically in network booting and therefore needed to be supported in order to implement a network bootloader.

- The old DHCP client duplicated code among the different DHCP request and reply handlers, thereby making it more difficult to fix other problems. Since all DHCP messages share the same basic format, it makes more sense to share parts of the code.

```
xsh$ netup ETH0
Trying DHCP on ETH0...
ETH0 is 192.168.1.72 with netmask 255.255.255.0 (gateway 192.168.1.1)
```

Figure 10: An example of using the Xinu shell to bring up a network interface using DHCP autoconfiguration. This only works if there is a DHCP server (not included with Embedded Xinu) running on a separate host on the network.

Consequently, I rewrote the DHCP client and added it to the current version of Embedded Xinu. It is now located in the `network/dhcpc` directory and is divided into the following functions, each of which is located in the same-named source file:

- `dhcpClient()`, which is the external interface to the DHCP client. As input, it requires the device descriptor for the network device on which to open the DHCP client (such as `ETH0`) and the number of seconds to attempt DHCP before timing out. As output, it produces the assigned IPv4 address, netmask, and gateway, as well as the bootfile name and IPv4 address of the "next" (TFTP) server if provided by the DHCP server.
- `dhcpSendRequest()`, which is used internally by `dhcpClient()` to send `DHCPDISCOVER` and `DHCPREQUEST` messages.
- `dhcpRecvReply()`, which is used internally by `dhcpClient()` to wait for an appropriate reply from a DHCP server.

Between requests and replies, the DHCP client stores its state in an instance of `struct dhcpData`. The code in the top-level `dhcpClient()` function resembles a state machine, as per the design of DHCP described in RFC 2131 [26].

In a network booting setup, the administrator of the DHCP server must configure the DHCP server to tell DHCP clients where to download the boot file (the new kernel). This information consists of the *file* (bootfile) and *siaddr* (next-server IPv4 address) fields that DHCP offers and acknowledgments contain. This information is returned by `dhcpClient()` if it was provided by the DHCP server. Consequently, it becomes available for the network bootloader to use in the next part of network bootloading, which uses TFTP to download the specified boot file from the specified server.

Although used as part of the standard network booting process, DHCP is designed for general IPv4 network autoconfiguration and is widely used for this purpose. Because of this, it was also pertinent to use `dhcpClient()` to implement DHCP support in the **netup** shell command. Example usage of this feature is shown in Figure 10.

Since Embedded Xinu's test suite is standalone and I did not implement a DHCP server for Embedded Xinu, it was not realistic to implement automated tests of the DHCP client. However, I tested the DHCP client manually by running the **netup** command as shown in Figure 10. To do this, I started the standard *dnsmasq* DHCP

server on another computer (running a Linux-based operating system), connected the Raspberry Pi (running Embedded Xinu) via Ethernet, and ran **netup** from the Embedded Xinu shell. This resulted in a successful DHCP transaction, and *dnsmasq* did not report any errors or warnings. I also tested that the DHCP client correctly timed out when no DHCP server responded. Furthermore, I tested that the DHCP client still worked correctly with its source code modified to drop random packets.

8.4 TFTP client

TFTP ("Trivial File Transfer Protocol") is a simple protocol for downloading and uploading files. It is an open protocol specified in RFC 1350 [47], although several extensions have been implemented [36][35][37]. TFTP is intended to be implemented over UDP and is designed to be small and easy to implement. A TFTP transfer begins with a request from a TFTP client to a TFTP server to read or write a file, which in typical implementations also serves to open a UDP connection. Following the establishment of a connection, the file data is transmitted over the connection in fixed-size blocks of 512 bytes. Each block is acknowledged by the receiving end, which provides some degree of resiliency against packet loss without incurring the complexity of running the protocol over TCP.

Although a TFTP client was available for a prior version of Embedded Xinu, it was not being source-compatible with the current version and the code quality seemed low. Consequently, I ended up implementing a new TFTP client from scratch. The resulting client is located in `network/tftp/` and can be programatically invoked using `tftpGetIntoBuffer()` or the more general `tftpGet()`. Both functions take in the name of the file to download and the network address of the server from which to download it. `tftpGetIntoBuffer()` stores the full downloaded file data into a dynamically allocated buffer, whereas `tftpGet()` provides the downloaded file data block-by-block to a caller-provided callback function. The client is robust to dropped packets. It conforms to RFC 1350 [47] but does not support any extensions of the base TFTP protocol defined in other RFCs [36][35][37].

TFTP is used in the standard networking booting process and is widely implemented by boot clients, which merits its discussion here. In addition, TFTP is very simple and would be easier to use as a basis for educational activities or assignments than other file transfer protocols such as FTP or HTTP, which further merits its inclusion in Embedded Xinu. However, since the current version of Embedded Xinu does not support a filesystem, other uses of the TFTP client are currently limited.

Since Embedded Xinu's test suite is standalone and I did not implement a TFTP server for Embedded Xinu, it was not realistic to implement automated tests of the TFTP client. However, after writing the client, I reviewed all the code for compliance with the TFTP specification [47]. Then, I tested the TFTP client manually by starting a standard TFTP server on another computer (running a Linux-based operating system), connecting the Raspberry Pi (running Embedded Xinu) via Ethernet, configuring the

network interfaces on each end, and downloading files to the Raspberry Pi via TFTP using a custom Embedded Xinu shell command. I tested that the client performed correctly when the specified file was found, when the specified file was not found, when the server did not respond, and with its source code modified to drop random packets. In addition, the TFTP client received further testing when I tested the integrated bootloader described in Section 8.6.

8.5 kexec: executing a new kernel in software

Once the bootloader has prepared the new kernel in memory, the final step is to transfer control to it. This task is routinely performed by special-purpose bootloaders, but it can also be implemented in more general-purpose operating systems. Linux, for example, supports a `kexec_load()` system call to load a new kernel, specified as one or more segments, into kernel memory. The loaded kernel can then be executed using the `reboot()` system call with a special flag, or executed automatically when a system crash occurs.

However, for the purpose of implementing a bootloader in Embedded Xinu primarily for the Raspberry Pi, a general interface like Linux's is not currently necessary. Instead, I added a `kexec()` function that simply takes in a single buffer of memory containing the new kernel and passes control to it:

```
syscall kexec(const void * kernel, uint size);
```

The actual implementation of `kexec()` for the Raspberry Pi port is more complicated than simply redirecting control flow to the memory address `kernel` (e.g. calling it as a function with no parameters). Since the firmware that boots off the memory card loads the first kernel at memory address `0x8000`, for consistency the kernel passed to `kexec()` is expected to also be linked to run at memory address `0x8000`. But since `0x8000` is already the memory address at which the currently executing kernel is loaded, `kexec()` must overwrite the current kernel with the new kernel. The trickiest part is doing this in such a way that the memory containing the instructions being executed to perform the copy is not itself overwritten.

The solution I came up with to correctly implement the `kexec()` semantics was to condense the code to copy the new kernel to memory address `0x8000` into a relocatable ARM machine code stub of fixed size (6 instructions long, to be precise). The disassembly of this stub is shown in Figure 11.

The stub takes as input the pointer to the new kernel in register `r0` and the number of 32-bit words in the new kernel image in register `r1`. For convenience, these correspond to the first two arguments to a procedure conforming to the ARM Procedure Call Standard[12]. However, unlike a normal procedure, the stub does not return when it has completed, but rather jumps directly to memory address `0x8000` to pass control to the new kernel. To prevent the stub from overwriting itself, `kexec()` copies it to memory address `0x7fe8` (just before the destination of the copy) before passing control to it.

```
 0:   e3a04902    mov     r4, #32768
 4:   e4903004    ldr     r3, [r0], #4
 8:   e4843004    str     r3, [r4], #4
 c:   e2511001    subs    r1, r1, #1
10:   1affffb    bne     4
14:   e3a0f902    mov     pc, #32768
```

Figure 11: Machine code stub used in `kexec()` implementation for XinuPi.

```
xsh$ kexec -n ETH0
Running DHCP on ETH0
Bringing up ETH0 as 192.168.1.54 with mask 255.255.255.0 (gateway 192.168.1.1)
Downloading bootfile "xinu.boot" from TFTP server 192.168.1.1
Executing new kernel (size=208832)
(Embedded Xinu) (arm-rpi) #9 (e@zzz) Sat Nov 23 20:25:29 CST 2013

...

Welcome to the wonderful world of Xinu!
xsh$
```

Figure 12: A demonstrating of the **kexec -n ETH0** shell command, which performs a network boot. This relies on DHCP and TFTP services configured on a separate computer on the network; see Figure 13 for details. The XinuPi Logo is omitted for space.

The `kexec()` implementation discussed here can be found in `system/platforms/arm-rpi/kexec.c`. I tested it by observing that it worked correctly when used in the integrated bootloader described in the next subsection.

8.6 Integrated bootloader

With an Ethernet driver, DHCP client, TFTP client, and `kexec()` function implemented, all the functionality needed for network booting is now available in Embedded Xinu. As a proof-of-concept demonstration of network booting, I implemented a **kexec** shell command that accepts the **-n NETDEV** option to perform a network boot over the specified network device, such as `ETH0`. An example of using this command is shown in Figure 12.

Although I implemented the proof-of-concept network bootloader as an interactive shell command, the operating system `main()` routine could be adjusted to run the same code automatically immediately after the operating system boots. This would result in a specialized image of Embedded Xinu that acts as an automated network bootloader.

```
$ ifconfig eth0 192.168.1.1/24 up
$ dnsmasq --interface=eth0                                          \
          --dhcp-range=192.168.1.2,192.168.1.254,255.255.255.0 \
          --dhcp-boot=xinu.boot                                     \
          --enable-tftp                                             \
          --tftp-root=/var/tftpboot
$ cp xinu.boot /var/tftpboot/xinu.boot
```

Figure 13: Example Linux/UNIX shell commands run on a separate computer connected via Ethernet to the Raspberry Pi in order to provide the network booting services necessary to reproduce the successful network boot shown in Figure 12. The **ifconfig** command sets a static network configuration. The **dnsmasq** command starts the DHCP and TFTP services. Its `--dhcp-boot` option specifies the filename to advertise to DHCP clients as the boot file to download using TFTP. The last command makes a XinuPi kernel available in the directory whose contents are made available via TFTP. These are example commands only, and other software can provide these same services.

Most of this new functionality is actually not dependent on the Raspberry Pi platform. However, for the Raspberry Pi, the firmware would start up the network bootloader stored on the memory card. The bootloader would then automatically download a different kernel from a server on the network and boot into it. This functionality is ideal for a laboratory environment like that used for previous ports of Embedded Xinu [17] [18]. If an equivalent lab with Raspberry Pis were to be deployed, then students would be able to remotely test bare-metal code, including Embedded Xinu kernels, on real Raspberry Pi backends without having to deal with the inconveniences of physically using the hardware, such as repeatedly copying the kernel to the memory card.

Chapter 9

Other Work

This section summarizes significant "other" work (i.e. work not primarily done by myself) towards porting Embedded Xinu to the Raspberry Pi or applying Embedded Xinu on the Raspberry Pi ("XinuPi") to education. It does not discuss other instructional operating systems in general, which is left for the last chapter.

9.1 Audio support

The Raspberry Pi cannot generate sound itself, but it can output a sound signal through the HDMI port, the GPIO pins, or the 3.5mm TRS connector. Currently, Embedded Xinu does not support sound on any platform, and it is arguably not an important feature to add. However, as an experimental project, Tyler Much attempted to figure out how to control the Raspberry Pi audio outputs from XinuPi. This section summarizes the results of his work.

Software running on the Raspberry Pi may send PCM[1] sound data over the GPIO pins by interfacing with the memory-mapped PCM module available on the BCM2835. Three separate signals may be controlled by software: the Bit Clock (`PCM_CLK`, GPIO pin 18, alternate function 0), Frame Sync (`PCM_FS`, GPIO pin 29, alternate function 2), and Serial Data Output (`PCM_DOUT`, GPIO pin 31, alternate function 2). Broadcom provides documentation for the PCM module [16]; however, the documentation does not specify certain key information, which had to be determined through empirical testing and finding alternate sources of information. More details can be found our WESE paper [14].

The code to interface with the BCM2835 PCM module is experimental and has not yet been added to the master branch of the Embedded Xinu repository.

[1] PCM stands for *pulse-code modulation* and is the standard form for digital audio in computers.

9.2 Graphics support

The Raspberry Pi does not, of course, include a graphical display itself, but it can output video through either RCA video or HDMI. In fact, the onboard VideoCore GPU, discussed in Section 3.2.2, is quite powerful and can be used for hardware-accelerated 2D or 3D graphics. Typical users of the Raspberry Pi running Linux attach a monitor and use it as the primary output device for human interaction. Although XinuPi is intended to be an embedded operating system and already has two other ways for humans to receive output from the operating system (serial port (Section 4.6) and Ethernet (Chapter 6)), basic graphics support is a very attractive feature to have that expands the potential user-base of Embedded Xinu, especially when combined with the support for USB keyboards (Chapter 7). Basic graphics support also demonstrates a feature that uses the VideoCore, which as discussed in Section 3.2.2 is an important part of the Raspberry Pi.

Graphics support for XinuPi was implemented by Farzeen Harunani. This section summarizes the results of her work.

Although the VideoCore supports APIs for hardware acceleration, for XinuPi we only desired basic graphics support using a framebuffer. A framebuffer is simply a region of memory containing pixel data which is periodically mirrored to the video display. After setting up a framebuffer, software can add or change graphics on the screen simply by changing the values of bytes in the framebuffer that represent pixels. The basic idea of a framebuffer is standard and is not specific to the BCM2835.

As mentioned in Section 3.2.2, the VideoCore is not well documented (publicly, at least), and this unfortunately extends to even the basic framebuffer support. To implement a framebuffer driver for XinuPi, Farzeen had to use various sources of information such as the C source code of a driver Broadcom provided for Linux and various forum postings and wiki pages.

To set up a framebuffer on the BCM2835, software running on the ARM must communicate with the VideoCore using a "mailbox system" that exists in hardware.[2] Like other devices on the BCM2835, the interface to the mailbox system is a set of memory-mapped registers located in the peripheral address range. Bidirectional messages sent through the mailbox system are limited to 28 bits. However, this is enough to store a 16-byte-aligned pointer to additional data.[3] This capability is used in the message to set up a framebuffer, which uses a structure to declare to the VideoCore the parameters needed for the framebuffer, such as the width and height of the physical display and the desired pixel depth. Following a successful framebuffer request by software running on the ARM, the VideoCore sends the address of the resulting framebuffer through the mailbox system.[4]

[2]The mailbox system has other uses as well; for example, it can be used for power management. In fact, it is needed to turn on the USB controller.

[3] As a "co-processor", the VideoCore can access the same memory that the ARM can.

[4]No cooperation from the dynamic memory allocator of the operating system running on the ARM is needed to allocate the framebuffer, since the VideoCore has a region of memory reserved for its use at

Following setup of the framebuffer, software running on the ARM can modify pixels on the screen simply by changing the values in the framebuffer memory. Farzeen built on this capability to implement text output using a hard-coded font, and on top of that a framebuffer device that conforms to the Xinu device model — that is, writes of character data to the device cause text to appear on the screen. Furthermore, she implemented functions to draw basic graphical shapes, and on top of that implemented an interactive turtle graphics program.

Although the turtle graphics program is interesting, from my point of view the framebuffer device is a much more useful feature, especially when paired with the USB keyboard support described in Chapter 7. More information about the graphics/framebuffer support for XinuPi can be found in our WESE paper [14] as well as in the source code itself. As of this writing, all the relevant code is included in the master branch of Embedded Xinu in the directory `device/framebuffer_rpi/`.

9.3 Classroom trials

Tyler was planning to work with Dr. Dennis Brylow to set up a Raspberry Pi laboratory environment at Marquette University for use in the operating systems course during the spring of 2014. The setup would be similar to previous laboratory environments used for Embedded Xinu [17][18] but would rely on low-cost, easy-to-setup Raspberry Pi hardware rather than Linksys routers requiring hardware modifications. If this laboratory environment is ever built, it would likely rely on the network bootloader that I implemented (described in Chapter 8). However, as of this writing I have heard no news regarding this, and the operating systems course appears to be being taught by another professor and is focusing on different topics.

Still, at least one other school has shown interest in the XinuPi work. A professor at Indiana University had students developing Xinu remotely on 32 Raspberry Pis during the fall of 2013. This used the XinuPi codebase discussed in this work as well as the *xinu-arm* codebase[32]. The differences between the XinuPi codebase and the *xinu-arm* codebase are discussed in the next section.

9.4 xinu-arm project

A project called *xinu-arm* overlaps somewhat with the XinuPi project. The xinu-arm code is publicly available via Git at `https://github.com/jetcom/xinu-arm` and is based on Embedded Xinu 2.01. It does not seem to be officially documented, but according to the version control logs, the project was started by several students at the Rochester Institute of Technology and has had several other contributors. Although the repository on Github is described as "Xinu arm port, for Raspberry Pi", the README file states a different goal: to "port Xinu to the ARM Versatile baseboard" (not the

boot time and can provide a buffer in this space to the ARM.

Raspberry Pi). Also, the Raspberry Pi code (platform *raspberry-pi*) is located in a separate branch of the repository, not in the default "master" branch, the latter of which contains two ARM platforms: *arm-qemu* and *fluke-arm*. Consequently, the xinu-arm project is perhaps best described as "an attempt to port Embedded Xinu to several ARM platforms, including the Raspberry Pi".

xinu-arm's Raspberry Pi support overlaps with the equivalent in our codebase ("XinuPi") primarily in the low-level details that I have categorized as part of the "core operating system" and covered in Chapter 4. This core functionality, such as preemptive multitasking and interrupt handling, is broadly equivalent between the two projects, although there are some differences. As an example, in Section 4.4, I noted that the threads implementation in xinu-arm's ARM platforms contained two specific bugs and had other issues; consequently, I wrote a new implementation of that functionality. I also identified logical errors in the organization of the xinu-arm code that make it less well-suited for educational use. For example, on the Raspberry Pi platform, xinu-arm implements `enable()` and `disable()` in a file named `vic.h`. Although `vic` stands for "Vectored Interrupt Controller", the Raspberry Pi does not, in fact, have a "Vectored Interrupt Controller", nor do `enable()` and `disable()` actually use the interrupt controller at all (see Section 4.2.1 versus Section 4.2.2).

Still, in Section 4.6 I noted that the PL011 UART driver in our codebase is, in fact, based on the one from xinu-arm, although that was in turn based on the NS16550 UART driver already included in Embedded Xinu. Indeed, because of Embedded Xinu's BSD-style license there is no reason why code from either project cannot be used in the other, unless one project starts releasing their changes under a different license.

Beyond the low-level details, the Raspberry Pi support in our codebase is much more advanced than the Raspberry Pi support in xinu-arm. Our codebase now supports USB and networking on the Raspberry Pi, and it is on these advanced features that I have spent almost all my time. Similarly, Farzeen spent most of her time on the graphics support, and Tyler spent most of his time on the audio and USB keyboard support. None of these features are included in the xinu-arm codebase.

I have also made many improvements to Embedded Xinu outside of those dealing strictly with the Raspberry Pi port, such as improving the documentation, fixing bugs in platform-independent code, and deduplicating code. In my opinion, some of these improvements have been more important than the Raspberry Pi work. For example, I have fixed various bugs unacceptable in any operating system, including incorrect implementations of trivial functions such as `memchr()` and a remotely-triggerable denial of service vulnerability in the networking stack.

Before March 2014, the main advantage the xinu-arm codebase had over our codebase was that only the xinu-arm codebase supported the *arm-qemu* virtual platform — that is, the QEMU open-source emulator software [13] simulating an ARM processor, specifically one located on an ARM Versatile Platform Baseboard. This platform may be of interest to instructors looking for a convenient, "hardware-free" solution for teaching operating systems using an ARM-based platform. QEMU also offers improved

debugging capabilities compared to real hardware.[5] Indeed, the absence of *arm-qemu* support from our codebase was apparently the main reason why the professor at Indiana University used the xinu-arm codebase in addition to ours.

However, in March 2014 I added *arm-qemu* support to our codebase. This was fairly easy because I had already factored out the Raspberry Pi code that would be common to all ARM platforms. Consequently, the main task I had to accomplish to achieve the port was to write drivers for the PL190 interrupt controller and SP804 timer present on the Versatile Platform Baseboard. Both devices are well documented by ARM Ltd. I did not borrow code from the *xinu-arm* codebase for various reasons, such as unclear code organization. Although I could discuss the *arm-qemu* work in more detail, it is not the focus of this thesis.

[5]QEMU allows single-stepping the processor to trace exactly what is happening. Although the Raspberry Pi can be debugged in a similar way using JTAG, it is difficult in part due to the explicit software configuration required to expose the JTAG signals over the GPIO pins.

Chapter 10

Conclusion and Related Work

This thesis has presented a port of the Embedded Xinu operating system to the Raspberry Pi in a project called "XinuPi". Embedded Xinu is an operating system targeted at modern embedded systems. It supports important features such as preemptive multitasking, interprocess communication, and networking, but it also has an instructional focus and features a relatively simple design. Porting Embedded Xinu to the Raspberry Pi has achieved two main goals. First, the project provides a convenient new platform for Embedded Xinu. Second, the project demonstrates an operating system running on the Raspberry Pi that is neither extremely complex nor extremely simple, thereby bridging a gap in the types of system-level software that run on the Pi.

Embedded Xinu is far from the only instructional operating system available. Anderson and Nguyen [9] surveyed instructional operating systems available as of 2005 that were suitable for use in undergraduate courses. Here I mention two they determined to be the most commonly used at that time.

Nachos (*Not Another Completely Heuristic Operating System*), developed at the University of California, Berkeley, can be provided to students as a skeleton to be completed by teams of two in a 15-week semester. Nachos is implemented in C++ or Java (depending on the version) and only runs on a simulator that emulates a MIPS processor. This distinguishes it from Embedded Xinu, which is implemented in C and runs on both simulators and real modern hardware.

The Minix operating system is a complete, fully-functional UNIX-like operating system based on a microkernel architecture. It is well-designed and is perhaps best known for influencing the early design of Linux. However, it includes about 30,000 lines of code and may be too complex for some students. (Unfortunately, Embedded Xinu currently has over 50,000 lines of code, but this counts all modules and all platform-specific code; by disabling or removing platforms or modules it can be stripped down fairly easily.)

Anderson and Nguyen also found that nearly half of institutions surveyed used no instructional operating system at all, but rather taught operating system concepts by having students use the UNIX API, write Java programs, use or study the Linux kernel in some way, or do something else. Although learning about operating system features

and writing programs that use them is very useful, this approach comes at the cost of missing the hands-on, low-level programming experience the "Xinu approach" offers. Furthermore, although the Linux kernel can make for a very good object of study, it is so complex and production-focused that teaching it well is very difficult and is outside the skills and knowledge of most instructors.

Slightly more recently, the xv6 operating system has been developed at MIT as a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6) [25]. Development started in 2006, about the same time Embedded Xinu development started. Unlike Embedded Xinu, xv6 supports multiprocessing, has better support for the x86 architecture, and supports a filesystem. A 2013 project has started porting it to the Raspberry Pi [62]; however, at this point, Embedded Xinu is farther along in its Raspberry Pi support.

Neither Embedded Xinu nor the Raspberry Pi have all the answers, but the combination of the two provides a framework for teaching operating systems at a time when they are becoming increasingly complex. Embedded Xinu can be used as a stepping-stone to understanding and working with full-blown, highly complex modern operating systems such as Linux; and despite some flaws, the Raspberry Pi currently provides an inexpensive and convenient platform for programming and computer science education, including operating systems development.

Although future work could involve adding new features to Embedded Xinu, the real strength of Embedded Xinu is its simplicity and understandability. Future work might focus on optimizing these aspects of the operating system while still maintaining relevance with new developments in the field.

Bibliography

[1] Raspberry Pi boot process. http://www.raspberrypi.org/phpBB3/viewtopic.php?f=63&t=6685, May 2012.

[2] Source code for bootcode.bin? http://www.raspberrypi.org/phpBB3/viewtopic.php?f=7&t=5170&start=10, April 2012.

[3] USB - the Elephant in our Room. http://www.raspberrypi.org/phpBB3/viewtopic.php?f=28&t=12097, July 2012.

[4] Bare metal. http://www.raspberrypi.org/phpBB3/viewforum.php?f=72, 2013.

[5] Embedded Xinu Wiki. http://xinu.mscs.mu.edu/Main_Page, 2013.

[6] RPi Hardware. http://elinux.org/RPi_Hardware, 2013.

[7] Community. http://www.raspberrypi.org/forum, 2014.

[8] James Adams, Gary Keall, Eben Upton, and Giles Edkins. Method and system for a shader processor with closely-coupled peripherals. Patent Application, 09 2011. US 2011/0227920 A1.

[9] Charles L. Anderson and Minh Nguyen. A survey of contemporary instructional operating systems for use in undergraduate courses. *J. Comput. Sci. Coll.*, 21(1):183190, October 2005.

[10] ARM Ltd. ARM architecture reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html, 2005.

[11] ARM Ltd. ARM1176JZF-S technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/index.html, 2009.

[12] ARM Ltd. Procedure call standard for the ARM architecture ABI r2.09. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042e/index.html, November 2012.

[13] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[14] Eric Biggers, Farzeen Harunani, Tyler Much, and Dennis Brylow. XinuPi: Porting a lightweight educational operating system to the Raspberry Pi. In *2013 Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*, October 2013.

[15] Broadcom. `arch/arm/mach-bcm2708/include/mach/platform.h`. Published via git: `https://github.com/raspberrypi/linux`, blob `e063a1ec36d92e69de64a7f07721a10d694a523f`, 2010.

[16] Broadcom Corporation. BCM2835 ARM peripherals. `http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, 2012.

[17] Dennis Brylow. An experimental laboratory environment for teaching embedded hardware systems. In *Proceedings of the 2007 workshop on Computer architecture education*, WCAE '07, page 4451, New York, NY, USA, 2007. ACM.

[18] Dennis Brylow. An experimental laboratory environment for teaching embedded operating systems. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, page 192196, New York, NY, USA, 2008. ACM.

[19] Dennis Brylow and Kyle 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.

[20] Alex Chadwick. Chadderz's simple USB driver for Raspberry Pi. Published via git: `https://github.com/Chadderz121/csud`, tree `ebdafcac3062a937164b704a3673c64ecc7e1eaa`, May 2013.

[21] Dom Cobley. config.txt. `http://www.raspberrypi.org/phpBB3/viewtopic.php?f=2&t=3042`, February 2012.

[22] Douglas E. Comer. *Operating System Design: the Xinu approach*. Pearson Education, 1984.

[23] Douglas E. Comer. The XINU page. `http://www.xinu.cs.purdue.edu/`, 2013.

[24] Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc., Microsoft Corporation, NEC Corporation, and Koninklijke

Philips Electronics N.V. Universal Serial Bus specification revision 2.0. http://www.usb.org/developers/docs/usb_20_070113.zip, April 2000.

[25] Russ Cox, Frans Kaashoek, and Robert Morris. Xv6, a simple Unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/2011/xv6.html, 2011.

[26] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFCs 3396, 4361, 5494, 6842.

[27] Raspberry Pi Foundation. Raspberry Pi Firmware. https://github.com/raspberrypi/firmware, 2013.

[28] hermanhermitage. Fun and games with the Videocoreiv Quad Processor Units. https://github.com/hermanhermitage/videocoreiv-qpu, 2013.

[29] hermanhermitage. Tools and information for the Broadcom Videocore IV (RaspberryPi). https://github.com/hermanhermitage/videocoreiv, 2013.

[30] hermanhermitage. VideoCore IV Programmers Manual. https://github.com/hermanhermitage/videocoreiv/wiki/VideoCore-IV-Programmers-Manual, November 2013.

[31] David Hollis, David Brownell, and additional contributors. USB network driver infrastructure. Published via git: git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git, blob 06ee82f557d45ba31b4847c187f57771ae2c73d2, 2013.

[32] David Larsen, Travis E. Brown, Zach Berger, David DiPaola, and Jeremy Brown. Xinu arm port, for Raspberry Pi. https://github.com/jetcom/xinu-arm, 2013.

[33] Nancy Lin, Steve Glendinning, and additional contributors. Driver for SMSC95XX USB 2.0 Ethernet Devices. Published via git: git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git, blobs 3f38ba868f6182152093e8efad0a864439c5cef0 and f360ee372554d5b1ce40c789aac44a6d066fc730, 2013.

[34] ARM Ltd. PrimeCell UART (PL011) technical reference manual r1p4. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf, 2005.

[35] G. Malkin and A. Harkin. TFTP Blocksize Option. RFC 2348 (Draft Standard), May 1998.

[36] G. Malkin and A. Harkin. TFTP Option Extension. RFC 2347 (Draft Standard), May 1998.

[37] G. Malkin and A. Harkin. TFTP Timeout Interval and Transfer Size Options. RFC 2349 (Draft Standard), May 1998.

[38] Adam B. Mallen and Dennis 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, page 161168, New York, NY, USA, 2010. ACM.

[39] Richard Miller. USB host driver for BCM2835: Synopsis DesignWare Core USB 2.0 OTG controller. `http://plan9.bell-labs.com/sources/plan9/sys/src/9/bcm/usbdwc.c`, 2012.

[40] Raspberry Pi Foundation. FAQs. `http://www.raspberrypi.org/faqs`.

[41] Raspberry Pi Foundation. Raspberry Pi Model B. `http://www.raspberrypi.org/wp-content/uploads/2011/07/RaspiModelB.png`, 2011.

[42] Raspberry Pi Foundation. About us. `http://www.raspberrypi.org/about`, 2013.

[43] Lubomir Rintel. Watchdog driver for Broadcom BCM2835. Published via git: `https://github.com/raspberrypi/linux`, blob `61566fc47f8441b9249054415acee30301e68b78`, 2013.

[44] RS Components. Raspberry Pi Getting Started Guide. `http://d4c027c89b30561298bd-484902fe60e1615dc83faa972a248000.r12.cf3.rackcdn.com/supporting_materials/Raspberry%20Pi%20Start%20Guide.pdf`, March 2012.

[45] Paul Ruth and Dennis Brylow. Teaching with Embedded Xinu: an inexpensive hands-on laboratory that promotes student engagement with operating systems, architecture, and networking curricula. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, page 3:13:2, New York, NY, USA, 2010. ACM.

[46] SMSC. LAN9512/LAN9512i: USB 2.0 Hub and 10/100 Ethernet Controller. `www.smsc.com/media/Downloads_Public/Data_Sheets/9512.pdf`, February 2012.

[47] K. Sollins. The TFTP Protocol (Revision 2). RFC 1350 (INTERNET STANDARD), July 1992. Updated by RFCs 1782, 1783, 1784, 1785, 2347, 2348, 2349.

[48] Theodore Sudol. Porting Lua to the XINU operating system. http://www.mscs.mu.edu/~brylow/reu/2012/Papers/Paper-2012-Sudol.pdf, 2013.

[49] Synopsys Inc. and additional contributors. Synopsys HS OTG linux software driver. Published via git: https://github.com/raspberrypi/linux, tree c4ed1c11fa14f3f6149575c1bce434f512869ff7, 2013.

[50] Technical Committee of SD Association. SD host controller simplified specification version 3.00. https://www.sdcard.org/downloads/pls/simplified_specs/archive/partA2_300.pdf, February 2011.

[51] Eben Upton. RISC OS for Raspberry Pi. http://www.raspberrypi.org/archives/2338, November 2012.

[52] Eben Upton. Android for all: Broadcom gives developers keys to the VideoCore kingdom. http://blog.broadcom.com/chip-design/android-for-all-broadcom-gives-developers-keys-to-the-videocore-king February 2014.

[53] Eben Upton. A birthday present from Broadcom. http://www.raspberrypi.org/archives/6299, March 2014.

[54] Ebon Upton. Model B revision 2.0 schematics. http://www.raspberrypi.org/archives/2233, October 2012.

[55] Liz Upton. Made in the UK! http://www.raspberrypi.org/archives/1925, September 2012.

[56] Liz Upton. Model B now ships with 512MB of RAM. http://www.raspberrypi.org/archives/2180, October 2012.

[57] Liz Upton. Wednesday grab bag. http://www.raspberrypi.org/archives/2665, December 2012.

[58] Liz Upton. FreeBSD is here! http://www.raspberrypi.org/archives/3094, January 2013.

[59] USB Implementors' Forum. Universal Serial Bus device class definition for Human Interface Devices version 1.11. http://www.usb.org/developers/devclass_docs/HID1_11.pdf, June 2001.

[60] USB Implementors' Forum. Universal Serial Bus HID usage tables version 1.11. http://www.usb.org/developers/devclass_docs/Hut1_11.pdf, June 2001.

[61] Dimitri van Heesch. Doxygen. http://www.stack.nl/~dimitri/doxygen/, 2013.

[62] Zhi Wang. xv6 for Raspberry Pi. https://code.google.com/p/xv6-rpi/, 2013.

[63] WG14. ISO/IEC 9899:1999, 1999.

[64] Michael Ziwisky and Dennis Brylow. BareMichael: a minimalistic bare-metal framework for the intel SCC. In Eric Noulard and Simon Vernhes, editors, *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, pages 66–71, Toulouse, France, July 2012. ONERA, The French Aerospace Lab.