Porting Darwin to the MV88F6281

ARMing the SnowLeopard.

Tristan Schaap
1269011

Apple Inc.
Platform Technologies Group

Delft University of Technology
Dept. of Computer Science

Committee:

Ir. B.R. Sodoyer
Dr. K. van der Meer

Preface

Due to innovative nature of this project, I have had to limit myself in the detail in which I describe my work. This means that this report will be lacking in such things as product specific- and classified information.

I would like to thank a few people who made it possible for me to successfully complete my internship at the Platform Technologies Group at Apple. First off, the people who made this internship possible, John Kelley, Ben Byer and my manager John Wright. Mike Smith, Tom Duffy and Anthony Yvanovich for helping me through the rough patches of this project. And the entirety of Core OS for making my stay an unforgettable experience.

Introduction

About the Platform Technologies Group
As it was described by a manager:

“We do the plumbing, if we do our jobs right, you never see it.”.

The Platform Technologies Group, a subdivision of the Core OS department, works on the embedded platforms that Apple maintains. Here, platforms are brought up and the embedded kernel and lower level support for the platforms is maintained.

What is Darwin?
Darwin is the lower half of the Mac OSX operating system. It includes the XNU kernel which is based on the Mach microkernel, and the userland.

What is the MV88F6281?
The MV88F6281 is an ARMv5te compatible processor from Marvell. It is based on their custom Sheeva core, which is designed to be like the ARM926EJ-S core from ARM. It has all the features you’d expect from a modern CPU, including several high speed SERDES lanes, a Harvard L1 cache, and 128kb of unified L2 cache.

The Project
The goal of this project is to get Darwin into a workable state on the MV88F6281 processor so that other teams can continue their work on this platform. The project has three major milestones:

1. Getting the buildsystem into shape, so that it can build the kernel and kexts.
2. Building and booting the kernel into single user mode.
3. Booting the system into multi user mode.

Because my project affects many other people, and it is encapsulated by a larger project. I joined the biweekly meeting for the encapsulating project. This would allow me to report back my progress, and get feedback from other engineers.

Summary

I worked in the Platform Technologies Group for 12 weeks, porting Darwin to the MV88F6281. The MV88F6281 is an ARMv5 compatible processor, with the custom Sheeva core at its heart. The goal of this project was to get Darwin building and booting into a full multi-user prompt. In order to achieve this goal, I set three milestones:

1. Get the system into a buildable state.
2. Boot Darwin into a single-user shell.
3. Boot Darwin into a full multi-user shell.

Because the project was to be built from scratch with the compiler from the new SnowLeopard operating system, several problems had to be addressed. These included problems with missing symbols, needing several kernel patches.

Getting Darwin to boot after getting it to build proved to be a major challenge, forking my project into two. After encountering a problem with an instruction set on the 6281 chip, we had to stop using the THUMB instruction set altogether. Because it was unclear at the time why the chip failed to correctly execute the THUMB instructions, I set out to research the problem in finer detail alongside the main project of porting Darwin.

After encountering several kernel bugs that stopped the kernel from booting, I achieved my milestone of a single user mode shell after about 8 weeks of work. After fixing several problems with both the kernel and the filesystem, I achieved my final goal of booting into a multi-user prompt.

Continuing my research into how and why the kernel failed to execute several instructions, I found that the problem lay not with the core itself, but rather with a poor implementation on the debug hardware. After completing all the milestones that had been set for me, I successfully completed the project.

Building a new platform

When I began this project, development for the MV88F6281 was being done with a train from a different ARM based project. This made sense, because both projects are ARM based, and thus share a lot of code, libraries and low-level platform support.

There are unfortunately multiple problems with this approach:

1. If one project made changes to the low-level platform support, these changes could have major repercussions on the other platform. I will discuss examples of this later.
2. When one of the projects is in the final stages of development of a release, no more changes to the lower level libraries are allowed. This means that the other project can also not make any changes to these lower level libraries. This poses serious problems when that project is in the early stages of development, where changes to the lower level libraries are far more prominent.

The solution to these problems is fairly simple. Create a new separate train for the MV88F6281. The Mac OSX buildsystem allows for what is called a Platform. A Platform is a complete replication of a root directory, and it contains a compiler, libraries and all the tools needed to build both userland and kernelland applications for Darwin. So one of the goals of this project is to create a Platform for the MV88FXXXX family.

Booting iBoot

iBoot is the bootloader of choice for embedded targets at Apple. My first task was to get iBoot building and booting on the MV88F6281. This involved setting up the JTAG environment needed to upload code to the target, as well as perform the initialization of the target. This includes setting up the SDRAM controller to allow access to the main memory banks.

The JTAG unit is a BDI3000, made by Abatron. One of its features is the setting of memory mapped registers, such as required by the SDRAM controller in the Sheeva core. Setting up the registers turned out to be trivial, using the W32 command in a BDI script, which will write to a memory mapped register. The needed SDRAM settings were glanced from the MV88F6281 datasheet, and the bootloader that was originally present on the device.

The actual process of building and booting iBoot on the MV88F6281 was fairly hassle free. Because iBoot is a bootloader, it has to be entirely self contained and cannot rely on external libraries. Even with only the ARMv5 compiler found in Snow Leopard, a simple:

make TARGET=airBoot PLATFORM=mv88f6281

Sufficed to build a binary image ready to be loaded using the BDI debugger.

Building the kernelcache

After acquiring the means to boot kernels on my device, the second task was to research the viability of using the vanilla ARM toolchain supplied with the new Snow Leopard operating system to build a kernelcache for the MV88F6281. This would identify the problems we would be facing when designing a completely new MV88FXXXX platform.

Snow Leopard comes with libraries for ARMv6 and ARMv7 targets, but has only a compiler for the ARMv5 target. This is not a problem for the goal we're trying to achieve, which is building the kernel. The kernel does not rely on any of the libraries, thus we should just be able to build it with nothing but a kernel.

make TARGETS="arm development mv88f6281" VERBOSE=yes

Does indeed build us an ARMv5 kernel. Building the kexts[1] we need in order to boot the system such as AppleARMPlatform build in much the same way.

The way embedded systems boot is with a kernelcache. This kernelcache contains all the kexts, and the kernel. These kexts and the kernel exist in a prelinked form, that is to say, no further linking needs to be performed by the kernel in order to use the kexts found in the kernelcache.
In order to perform this prelinking, a kernelcache builder needs to be used. This will invoke the kext linker, kxld, to perform the linking of the kexts into the kernel. The kernelcache builder will then build the kernelcache out of these kexts and the kernel.

When prelinking the kexts, kxld will try to resolve all the imported symbols using symbols exported by other kexts. Most notably, the System.kext, which is the kext associated with the kernel. This is where I ran into the first problem. Building the kernelcache for the first time produced several errors.

```
kxld[com.apple.driver.AppleARMPlatform]: The following symbols are unresolved for this kext:
kxld[com.apple.driver.AppleARMPlatform]:     .constructors_used
kxld[com.apple.driver.AppleARMPlatform]:     .destructors_used
kxld[com.apple.driver.AppleARMPlatform]:     __start
kxld[com.apple.driver.AppleARMPlatform]:     __stop
```

These are C++ related symbols, and should have been defined in the kernel, and more specifically, in libkmod. libkmod is the library that all kernel modules use to initialize themselves and hook into the kernel. After much searching, it turned out that a #define in the kernel was the culprit.

---

[1] Kexts, or kernel extensions are what drivers are called.

In the latest version of the SnowLeopard operating system, the C++ runtime had been changed for the x86_64 platform. This involved relocation several symbols to another part of the kernel. Unfortunately, the way this was implemented in the header was

#if __i386__ ll __ppc__

Which will indeed cause the desired behavior, the symbols will no longer be defined for x86_64. It should however been implemented as

#if !__i686__

Which will once again define the symbols for __ARM__. Reproducing this fix in several other header files and adding some of these symbols to the symbol_export list allowed me to fully prelink the kernel and kexts, producing a valid kernelcache.

Booting the kernel

After building the kernelcache, the next step was to try and boot the kernelcache. My initial attempt at booting the kernelcache landed me somewhere halfway through the boot. The processor hung, and no specific error was given. Using gdb through my BDI JTAG debugger, I traced back the hang to one specific line of code.

```
record_startup_extensions_function();
```

Right after figuring this out I attended one of our bi-weekly status meetings. I mentioned this particular problem at the meeting, and one of the engineers present explained to me that this was a function that was used for dynamically linking in kexts into the kernel. Because we are using a kernelcache, we don't have to dynamically load kexts. In fact, we can't because we're missing the kxld executable. Because the function was of no use to us, I simply commented it out.

Commenting out that function allowed the system to boot slightly further, right up to the point where it was loading kexts into memory and pairing them to actual devices. The system now came to a halt with a more explicit error, and a kernel panic.

```
panic(cpu 0 caller 0xc0230569): "Unable to find driver for this platform: \"AppleARM\".\n"@/
Users/src/xnu-trunk/iokit/Kernel/IOPlatformExpert.cpp:1389
```

This panic means that the kernel is unable to find the kext for the AppleARM platform. More specifically, it is missing the AppleARM PlatformExpert. This is the kext that has all the information on the underlying system, and functions to modify and access it. Information includes the operation of the Interrupt Controller, and other systems that are critical to operation.

The kernelcache however did include this AppleARMPlatform kext. This problem led me on a wild goose chase around the entire buildsystem. The buildsystem was a likely culprit, because it was a largely untested system. After replacing several key items of the buildsystem, including the compiler, kernelcachebuilder, kxld, and several libraries I was no further to my goal. The only thing that had not been researched was the kernel itself. Once again using gdb and several debug printfs I located the function that was supposed to load these drivers into the kernel.

```
record_startup_extensions_function();
```

Turned out that the function that according to the engineer only dynamically loaded kexts was also responsible for loading prelinked kexts into the kernel. I uncommented the function. I had gained valuable knowledge as to how the buildsystem worked, having tore it apart pretty thoroughly, but no closer to my goal of booting Darwin.

Debugging using JTAG

Setting up my JTAG unit to catch all the processor exception vectors, I started to walk through the offending function. Using a technique called singlestepping, I was able to narrow down the problem to a single snippet of assembly, shown below.

```
c00814be            4640        mov     r0, r8
c00814c0            990c        ldr     r1, [sp, #48]
c00814c2        f005eaac        blx     0xc0086a1c
c00814c6            e79f        b.n     0xc0081408
c00814c8            4803        ldr     r0, [pc, #12]
c00814ca        f7a0f823        bl      0xc0021514
c00814ce            e781        b.n     0xc00813d4
```

Single stepping this code showed the problem. The JTAG unit caught an exception while trying to execute the code at 0xC00814C2.

```
Core number       : 0
Core state        : debug mode (ARM)
Debug entry cause : Exception (UNDEFINED)
Current PC        : 0xc00814c0
Current CPSR      : 0x60000093 (Supervisor)
```

After carefully inspecting the offending code, it was deemed that this was actually valid THUMB code. The target of this BLX instruction, which jumps to the specified address and changes into ARM mode, was also valid code. This problem turned out to be insurmountable, and after several meetings we decided to entirely drop half of the instruction set. The suspicion was that a silicon bug in the Sheeva core made the unit fail on certain instances of the BLX instruction. This suspicion was enhanced by the fact that a previous Sheeva based device, the MV88F5281 exhibited similar behaviour and that unit had been proven to have a silicon bug. This bug was supposed to have been fixed in the newer MV88F6281 but it was possible that they introduced a new bug while trying to fix the old one. Because the project needed to be completed anyway, the decision to drop the THUMB instruction set was made. I did continue to investigate, and finally solve, the problem with the THUMB instruction set. I will discuss this in a later chapter.

This means that we now had to re-engineer the platform to not compile any THUMB code, but rather only ARM code. Fortunately, GCC contains a flag to disable all thumb code, -mno-thumb. This flag was used throughout the Platform, and no more THUMB was produced. This however was not the solution to all our problems, as the unit still did not boot, and still gave me no obvious warning when booting. However, singlestepping through the code now no longer gave me an undefined instruction exception, so I was able to locate the real problem.

Threadpointers

In the XNU kernel a pointer that points to the current threads information struct, or current thread-pointer, is something that is always kept handy. In order to do that, it is always stored in a specific place. On ARMv5, it was decided that in the kernel the register r9 was to be used for this purpose. Later revisions of the ARM specification, such as ARMv6 and ARMv7, provided a dedicated register to be used, stored in a CP15 register. But because all ARM platforms share a lot of their code, r9 should still not be touched in ARMv6 and ARMv7. This is only an issue in hand rolled assembly, because GCC has had rules added to it to prevent it from using the r9 register.

But because ARMv5 is a platform that was not being actively developed for at Apple, the code for this platform was not being exercised. This meant that it was possible for a function to trash the r9 register with it going unnoticed. This because when the code was being run on either ARMv6 or ARMv7 the current thread pointer would be restored from the dedicated CP15 register before use.

After spending considerable time tracing my steps through several assembly files, the problem was finally identified in the OSAtomicAdd function, which was used by copypv, which in turn was used by the function loading the kexts into memory to copy the kexts from physical to virtual memory. The problem meant that after calling the OSAtomicAdd function, the value in the r9 register would not be pointing to the current thread struct, but instead to a random piece of memory. Once the kernel used any of the values in this struct, it would generate a data abort. Due to the r9 pointer being corrupted, the data abort handler itself would generate an abort when trying to print out the panic, thus creating an infinite loop of aborts and hanging the kernel.

Even though the problem was hard to track down, the fix was fairly trivial. Replacing the use of r9 with another register allowed the system to continue booting. The system actually continued to boot up to the point where it required a filesystem. Setting up an NFS server with a filesystem borrowed from another ARM based unit, I achieved one of the milestones of the project, a single user prompt.

Multiuser mode

Achieving the single user milestone presented me with the next challenge, achieving multi-user mode. The difference between multi-user mode and single user mode is fairly small.

1. In single user mode launchd does not initialize any services.
2. Multi-user mode provides you with a prompt
3. In single user mode launchd does not attempt to open a pipe to allow launctl to talk to it.

Even though the difference between single and multi-user mode is fairly small, it was large enough to create problems booting into it. After booting into multi-user mode, the system would hang. Once again without any sort of descriptive error. This time my JTAG unit nor GDB was of any help, as the unit did not fatally crash but rather hung in a function waiting to receive a message from an IPC socket. However, knowing that we had a single-user prompt, it had to be something that was only being done in multi-user mode, drastically reducing the number of problems.

Through a process of elimination, one shell script was found to be the very unlikely culprit. /etc/.profile is the systemwide default init script for sh. It does a minimal setup of environment variables and other things needed to run the shell. The offending line in the script, by another process of elimination, was found to be eval `/usr/libexec/path_helper -s`. Booting back into single-user mode and trying all the elements separately:

eval true
/usr/libexec/path_helper -s
`echo true`

Showed that the problem was in using the back ticks. In SH, back ticks mean “execute this and replace myself with the output”. So for `echo true`, SH would execute echo true, receive the output ‘true’ and execute that. After some experimenting, we found that the simplest use-case which crashed the system was the command line whoami &. This will execute the whoami command in the background. This generated the following output:

```
bash-3.2# whoami &
root
```

This meant that the process was actually being executed before the system died. At one point I accidentally resized the SSH window I was using to debug this issue. Resizing the window made the unit crash. This fact combined with the fact that backgrounding a task crashed the unit led us to believe that the problem had to do with signals. The act of backgrounding a task spawns a child process, which then terminates. This sends a SIGCHLD to the parent task. When a window is resized, this sends a SIGWINCH signal to the process.

After determining that the signal handler was the culprit, I started looking through the second level SWI handler which is written in ARM assembler. This handler at one point calls the function set_cthread_ptr. Because of the way this function is designed the loading of the arguments happens about a hundred lines away from the actual function call, unless you're running on ARMv6 or ARMv7, in which case the argument will load right before calling the function. This again because they use the special purpose register to store the current thread pointer.

Because of this separation of loading the arguments and calling the function, the r0 register which held the arguments and is generally considered to be a register for momentary use got used in some revision of the kernel. This broke ARMv5 support for signals. Changing the register used from r0 to a non-used register such as r10 fixed the problem. Doing this allowed signals to work once again on ARMv5 systems:

```
-sh-3.2# whoami &
[1] 4
root
-sh-3.2# true
-sh-3.2#
```

This however didn't fix the multi-user problem. However, adding several debug printfs to the launchd deamon showed a problem where it was unable to write to /tmp/launchd. Launchd will write a named pipe to that location, in order for launchctl to communicate with it. With that named pipe missing, launchctl could not tell launchd to start all the services. After deliberating with the build engineer responsible for the project, he decided to add an extra project to the build that would recreate the /tmp directory that had been removed in an earlier erase-empty-directories step. With this fix in place, the system booted to a full multi-user mode, achieving the final milestone in the project.

THUMBs down

As was quickly glossed over in the ‘Booting the kernel’ chapter, we encountered an issue with the THUMB instruction set in the MV88F6281’s Sheeva core that forced us to abandon this instruction set, and move to an entirely ARM kernel and userland. Because it was important to both Apple and Marvell to understand this problem, I continued to investigate it alongside my porting work.

The assumption that I was working off was that the THUMB instruction set was broken. So I set out to prove this. After miraculously finding that the kernel got past the earlier problem of the copypv, I tried writing some of my own assembler to exercise the problem. This assembler can be found in Appendix A. I singlestepped this code and successfully exercised the problem.

```
         User     FIQ     Superv   Abort     IRQ      Undef
GPR00: 00000000 00000000 00000000 00000000 00000000 00000000
GPR01: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR02: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR03: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR04: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR05: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR06: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR07: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR08: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR09: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR10: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR11: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR12: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR13: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
GPR14: c0086396 c0086396 c0086396 c0086396 c0086396 c0086396
PC   : c0086380
CPSR : c00863b6
SPSR :          c0086396 c0086396 c0086396 c0086396 c0086396
```

The above trace shows one of the unfortunate symptoms of the problem, bogus information in all the hardware register. All the registers are returning close to the same value. This made the debugging process all that much harder, because I had no idea what state the processor was really in after executing this instruction. I discussed this with another engineer. Because the symptoms made regular debugging nigh impossible, he put me in contact with Jenning Chee, one of Marvell’s lead hardware designers.

After explaining the problem to Jenning, he relayed my test results and code to the hardware design team in Israel. This team tried to unsuccessfully replicate my results on identical hardware. Replacing my unit with a new one however got me the same results as I got earlier. The only difference between my setup and that of the team in Israel was that they were using a Lauterbach JTAG unit, instead of my BDI3000 JTAG unit. After this confusing result, I tried executing my code once again, without the BDI3000 JTAG unit attached. It worked.

Because the only variable that had changed in the equation was the JTAG debugger, it was now the main suspect. After some more experiments, this was found to be true.

Even though the Sheeva core is most likely not the problem, the THUMB instruction set was not put back in the MV88F6281 platform. This because the BDI is the most used JTAG debugger at Apple, and changing this unit would be much more of a hassle than disabling the THUMB instruction-set on the platform.

Because this issue is still under investigation and certain details of the debug module and process contain information which is under Non Disclosure Agreement, I cannot speculate as to the cause of this problem.

Conclusion

In the 12 weeks that I spent working on this project I learned more about embedded systems, buildsystems and debugging than I could have ever hoped for. Having had the opportunity to go deep down into the crevices of a complicated embedded devices in my research into the THUMB issue has taught me more about the inner workings of an embedded device than any lecture could have ever done.

Also having the chance to deliberate with engineers on both the Apple side, and the side of a vendor has strengthened my ability to communicate with other engineers at a technical level and has given me insights into the operations of a major company. Attending the bi-weekly project meetings showed me how a large project such as this is handled on an operational level, and a human resources level.

I faced several technical difficulties during this project, all of which can be roughly subdivided into three categories:

1. Buildsystem. Having to create a buildsystem from the ground up, including the filesystem and kernelcache.
2. Stale kernel source. Because the ARMv5 branch of XNU had not been exercised in a long time, bugs snuck in.
3. JTAG debugger. Due to issues with the JTAG debugger, an entire instruction set could not be used.

Having to deal with all these issues showed me how platform bringup worked from a very low level where I had to get the system to compile, to a fairly high level, where I had to debug issues on a filesystem level. Achieving all of the milestones that I set out to achieve, I successfully completed the project.

Future Work

Getting the unit to boot to multi-user mode is a big step forward, but it is not quite ready. Several things have to be done before this product is ready to ship.

L2 Cache
The L2 cache in the MV88F6281 is Virtually Index and Virtually Tagged, or VIVT. XNU was not designed to handle this. Several changes have to be made to the kernel in order to exploit the advantages of the 128kb unified L2 cache.

Drivers
The hardware has to have several more drivers ported to it, to fully utilize the potential.

Userland
The bare minimum userland that has been ported to the platform is not enough to perform the tasks the unit needs to perform. Several applications will have to be written or ported from other platforms.

Glossary

Term	Definition
ARMv5	The fifth incarnation of the ARM specification.
Breakpoint	A preset point at which the execution of the processor will be halted, to allow for debugging.
CP15	Coprocessor 15 in several ARM processors is used to control a whole range of system functionality such as the memory management unit.
Exception Vector	The ARMv5 specification calls for several exception vectors to be in place. These vectors are placed at a fixed spot in memory and are called whenever an exception such as an undefined instruction occurs.
GDB	The GNU debugger is used to debug applications on a high level
Interworking	Interworking on THUMB enabled platforms means switching between the ARM and THUMB instruction sets.
JTAG	A port that allows control over the CPU on the lowest level. It allows you to halt the CPU, and analyze its current state.
Kernelcache	A kernelcache is a binary blob containing the kernel and all the necessary prelinked kexts.
Kext	Short for kernel extension, kexts are drivers.
Singlestepping	The technique of executing one instruction for every key press by the user
SWI	Software Interrupt. An instruction that allows running code to generate an interrupt. Commonly used for syscalls.
THUMB	An extension to the ARM instruction set, THUMB is a completely separate instruction set. It is 16 bit instead of 32 bit, allowing for space savings up to 40%.

References

- Marvell, *88F6281 Hardware Specifications document*, December 1, 2008*.
- Marvell, *88F6180, 88F6190, 88F6192 and 88F6281 Functional Specifications document*, December 1, 2008*.
- Marvell, *Document Changes and Updates 88F6281 Hardware Specifications, Revision E and 88F6180, 88F6190, 88F6192 and 88F6281 Functional Specifications, Revision C,* January 28, 2009*.
- ARM, *ARMv5 Architecture Reference document*, 2005.
- Marvell, *Feroceon L2 Addendum*, October 23, 2007*.
- Apple, *internal kernel documentation*.*
- Apple, *internal iBoot documentation*.*
- Marvell, *Preliminary Feroceon Core datasheet,* January 30, 2008*.

* Document only available under NDA

Appendix A

```
/*
 * Copyright (C) 2007 Apple Inc. All rights reserved.
 * Copyright (C) 2006 Apple Computer, Inc. All rights reserved.
 *
 * This document is the property of Apple Inc.
 * It is considered confidential and proprietary.
 *
 * This document may not be reproduced or transmitted in any form,
 * in whole or in part, without the express written permission of
 * Apple Inc.
 */

        .text
        .align 4
        .arm
        .global _do_crash_asm
_do_crash_asm:
        b .
        mrs r0, CPSR
        bic r0, r0, #0x0F
        orr r0, r0, #0xC3
        msr CPSR, r0

        mov r0, lr
        blx do_crash_asm_thumb_aligned
        blx do_crash_asm_thumb_non_aligned
        bx r0

        .thumb
        .align 4
do_crash_asm_thumb_aligned:
        mov r8, r8
        mov r1, lr
        blx do_crash_asm_arm
        bx r1

        .align 4
do_crash_asm_thumb_non_aligned:
        mov r1, lr
        blx do_crash_asm_arm
        bx r1

        .arm
        .align 4
do_crash_asm_arm:
        mov r0, r0
        bx lr
```

Appendix B

Skills

This project required a high level of understanding of methods of debugging embedded systems, as well as an affinity with both C and assembly coding. Having already worked on several projects requiring such skills, I was confident in my ability to overcome any difficulties that would undoubtably arise. In order to show my preparation for the task at hand, I will discuss two of the projects that have given me the skills necessary to complete the work on the MV88F6281.

Prex

Having always possessed an interest in all things embedded, I came across an obscure Chinese made handheld device. Opening the device showed me a processor marked as an SPMP3050. After some research I determined this was an ARM926EJ-S based processor, much like the one the Sheeva core in the MV88F6281 is based on. Unfortunately, no datasheets were available for this specific processor. So in order to make use of the various peripherals of the device, such as the LCD screen and serial port, a certain amount of reverse engineering of the existing firmware was needed.

Reverse engineering is the process of taking assembled binary executable code, and reversing the process of assembly. This yields a long listing of assembly instructions, equivalent to the original binary. In order to read these specific instructions, one must have intimate knowledge of the underlying system. Having worked with several ARM systems before, I started reverse engineering the memory map of the device.

Reading a considerable amount of ARM assembly, I figured out how several of the peripherals worked. Once I had the basics of the device figured out, I started work on porting an RTOS (Real Time Operating System) to the platform. After much deliberation, it was decided this should be Prex. Lacking in experience regarding operating systems, I researched several aspects of the Prex kernel.

With new insights gained from reading both sourcecode and documentation, I started work on porting the bootloader and base Prex kernel to the device. Writing low-level device drivers involves a mixture of C and ARM assembly. After completing several of the device drivers, among which the serial port, LCD and GPIO interfaces, we had a fully working Prex kernel running on the device.

Prex - Conclusion

I gained several insights into the inner workings of an operating system, something which would be highly valuable when working on the MV88F6281. I also honed my reverse engineering skills, which would come in handy when searching for possible bugs.

Development board

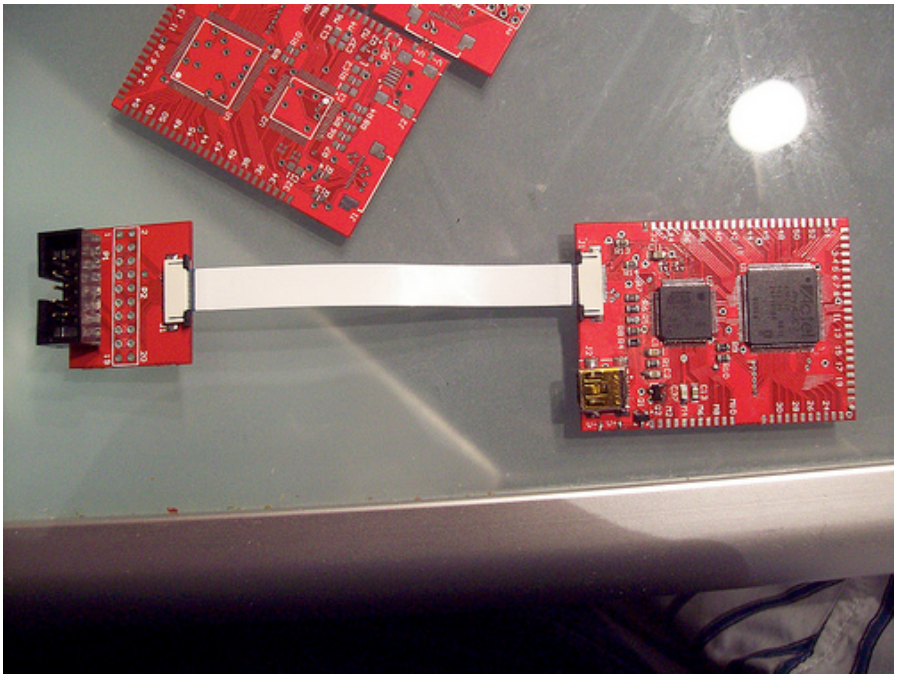
For another project, I required a small development board with a high amount of high speed general purpose IO pins. Rooting around my drawers I could only find larger development boards. These did not meet the requirements for the project, which called for a small development board.

Scouring the internet for a device that would meet the requirements, I came up empty. This was a niche that was apparently unfilled, so I decided to fill it myself, by designing my own development board.
Drawing up a few specifications on my whiteboard, I came up with the following:

- Must be able to sample and write at speeds of up to 166MHz
- Must be able to interface to a PC over USB
- Must be flexible

After much deliberation, I decided upon the components for the board. A highspeed FPGA would be used for the sampling of the GPIOs, and a microcontroller would interface with the FPGA and provide USB capabilities. Having an affinity with Actel FPGAs and ARM microcontrollers, I chose the A3P250-VQ100 as the FPGA, and the AT91SAM7S128 microcontroller by Atmel.

The PCB design was done in Altium designer, and the parts were ordered from Mouser. I ordered the fabrication of the PCBs from Myro PCB in Shenzen, China.
After all the parts arrived, I assembled an early prototype.



Debugging the hardware took some time, but once all the components were enumerating on their respective busses (FPGA on JTAG, microcontroller on both JTAG and USB), I ported an operating system called eCos to the microcontroller.

Development board - Conclusion

Developing my own development board from the ground up gave me valuable insights into how hardware is designed, and how to properly read datasheets to get the required information out of it. Porting yet another operating system to a microcontroller strengthened my knowledge of operating systems even further. Using JTAG to debug the ARM7TDMI processor gave me experience in the low-level debugging field.

Process

Quality assurance

Working on a large project such as the XNU kernel, there has to be a form of quality assurance in order to keep the project building. Unfortunately, unit-tests as they are commonly used on traditional software projects don't work on the lowlevel code that is a kernel. Therefor, another form of quality assurance is used on the XNU project as well as several other embedded projects at Apple.

The procedure to getting your changes mainlined and included into the trunk is to file a CCC to the bug review board. The CCC I filed for the inclusion of my first patch is included here.

```
- Location of Diffs: Attached to Radar under filename pr-7169646-1493.patch
- Reviewed by:John Doe
- Branch: svn+ssh://src.apple.com/svn/xnu/branches/PR-7169646-1493
- B&I Project Name(s): XNU
- Project(s) that need to be rebuilt: XNU
- Integrator: Linda
- Built for architectures: ARMv5
- Build Built On: SnowLeopard 10A411
- Build Tested On: XNU trunk rev. 65168
- Machines Tested On: K30Alt

- Tested by: Tristan Schaap
- Testing instructions:

Build XNU, verify XNU boots past kext loading.

---Embedded-Only Submission Information---
- Risk Level: None
- Risk Details: None that I know of, only affects ARMv5 targets
- Review Details: None.
- Security Details: None that I know of.
```

It lists all the information needed to process the change, including the name of the person who peer-reviewed the change. All changes must be peer-reviewed, in order to reduce the chance of a bug or regression sneaking into the kernel. It also lists the person who will be integrating the change. Because so many people work on the XNU kernel, it is impossible and unwise to have each person integrate their own changes. Because of this, integrators will accumulate a certain amount of patches and then commit them to the trunk of the project.

This CCC is submitted to the BRB (Bug Review Board), to check if there are any problems. If not, they are forwarded to the integrator who will integrate you changes.

Reflection

Working as part of a large team, for a large company and on a large project has given me food for thought. Even though the secrecy that is inherent to working for Apple shapes the company and allows it to create some of the most innovative products on the market, it has a downside in that you aren't allowed to talk about possible difficulties you're facing with people that do not have clearance on your project. People that might have valuable insights or even solutions.

Because a large portion of my project was about porting old code, I had to interact a lot with people who had written this code. Luckily, most of these people knew about my project and I had little problem talking to them. However on a few occasions I had to be very cryptic about what I wanted or needed, impeding my progress.

This progress was slow at times, resulting in me being very frustrated. This is inherent to doing complicated low-level work, where progress is often achieved in bursts rather than a constant increase. Looking back on my project, I feel that I might have benefitted from a more structured way of working. This 'chaos' can partly be attributed to the fact that even though I had previous operating system experience, I was unfamiliar with the Darwin way of handling things. Someone more experienced with Darwin would probably have been able to know instinctively where the bug was most likely to be.

If I were to redo this project I would have probably tried to work closer with the other engineers available to me, and do more teamwork. I did learn a lot more about the XNU kernel trying to figure out all the problems myself, but I would have been able to finish my project quicker if I had asked for more help.

I don't believe my preparation was lacking, I had little to no problems adjusting to the MacOSX development platform, drawing on my experience with the UNIX commandline. Further drawing on my skills with ARM assembly and the C programming language, I was able to quickly orient myself and start with the task at hand.

In conclusion, even though I had a few hiccups and several unforeseen problems arose during the project, I am satisfied with the way it turned out. Several details could have been improved upon, but I have learned from these mistakes, which means they were not made in vain.

Appendix C

Plan of Approach

Porting Darwin to the MV88F6281

Plan of Approach

Tristan Schaap
1269011

Apple Inc.
Platform Support Group

Delft University of Technology
Dept. of Computer Science

Committee:

Ir. B.R. Sodoyer
Dr. K. van der Meer

Preface

Because the planning of this project is largely dependent on intermediary results, some parts of this planning may be intentionally left vague. Also, because the project in question is one that is highly innovative, several product-specific and confidential details have been left out.

This document describes the requirements of the project. The goal of the project is to achieve a fully working Darwin multi-user environment on the MV88F6281 processor by Marvell. In order to achieve this, several milestones will be decided upon, and a rough planning will be made.

Plan of approach

Summary

For this internship at Apple, I will be porting Darwin to the MV88F6281, an ARMv5 compatible CPU based on the custom Marvell Sheeva core. Several steps will need to be taken to achieve the final goal of being able to boot into a multiuser prompt. Due to the unpredictability of the steps, only rough estimates will be made as to how long a given step will take. Only one deadline will be implemented. This deadline is for the final milestone, booting into a multiuser prompt. This deadline will be set to the 25th of September.

I will start my project in Cupertino, California on the 6th of July and will be working in 2 Infinite Loop, on the 4th floor until the 25th of September. There will be bi-weekly meetings with the team, allowing for status updates and discussing further steps.

Introduction

Darwin is the combination of the XNU kernel, based on the Mach microkernel, and the accompanying userland. Darwin is in use on both the desktop operating system MacOSX, as well as the embedded iPhone OS. With this in mind, it makes sense for any new platform to use Darwin as well, as this focusses all engineering effort on maintaining and improving Darwin, rather than spreading the effort out over several operating systems and thus diluting the effectiveness.

The intent of this project is to port Darwin to the MV88F6281 processor, and allowing it to be blessed with the multitude of network protocols and applications already developed for the Darwin operating environment. In the process of porting Darwin, several design decision will have to be made on the fly, making the planning a slightly volatile and dynamic one. Bi-weekly meetings with the project members and management will resolve any issues that arise, and further planning will be decided based on these issues.

Even though the planning might be fairly volatile, a few milestones will be decided upon, allowing me to measure my progress through the project.

Contract

Project goal

The goal of this project is to port Darwin to the MV88F6281. The expected outcome of this project is a fully working multi-user environment on the target hardware, allowing for further development of the target hardware.

Requirements

The port of Darwin must be developed in the MacOSX development environment, using the currently unreleased SnowLeopard operating system. All development will be done in C, or ARM assembly and debugging will be performed with in-house tools.

Darwin must boot entirely to the multi-user prompt, however it is not necessary to have complete hardware support. Drivers will be developed at a later stage in the project.

Planning

As mentioned earlier, the planning of this project is by no means a simple task. Because deciding the best course of action depends on the intermediary results and a fixed planning does not fit within the philosophy of the Platform Support Group. I decided to go with the development method as it exists within the Platform Support Group, which is to break the project up into several milestones, and give a rough estimate as to how long each milestone will take. This allows for a certain amount of stretch in your planning, as embedded development is unlike normal development. Problems present themselves in a multitude of ways, and because you are often working with prototype hardware, it is unclear whether the issue is hardware, software or a combination of both.

Three major milestones exist within the project:

1. Getting the XNU kernel, needed kernel extensions, and userland into a buildable state.
2. Being able to boot Darwin into a single user mode
3. Being able to boot Darwin into a multi user mode

Achieving the first milestone should be reachable within 3 weeks, the second milestone will take about 4 weeks, and the final milestone should take about 3 weeks again. This leaves about 2 weeks of leeway to account for the inevitable problems I will encounter.

Quality assurance

Progress shall be evaluated at the bi-weekly meetings.