Reflective, Object Oriented Operating System Kernel Design and Development for the ARM Architecture

George Morgan

March 2012

ABSTRACT

The development of modern operating system software to accompany the Advanced RISC Machine (ARM) microprocessor is falling exponentially behind the evolution of the architecture itself. Embedded systems utilizing the infrastructure of the ARM processor are in a constant cycle of redesign, whereas the system software developed for these platforms is not. To provide these platforms with an operating system, a developer must port an operating system from one processor core to the next. Developers are responsible for modifying the framework of the kernel to accommodate for the remodel of each new microprocessor revision. With each consecutive refactorization of the target operating system's framework comes a decrease in efficiency in the dependent kernel-space and user-space applications.

This study in operating system design and development focuses on a series of procedures that explore new processes and techniques of embedded system software development for the ARM architecture. Throughout the project, a new method of kernel processing was developed by applying a reflective, object-oriented kernel environment into the operating system as well as implementing a new method of low-level firmware design.

This project demonstrated that when comparing the results between the developed operating system to others that do not utilize a reflective and object oriented kernel, the Instructions Per Cycle (IPC) are noticeably higher. These results prove that this specific approach to kernel processing dramatically improves the efficiency of the operating system on each microprocessor core without the need for a framework refactorization.

STATEMENT OF THE PROBLEM

Since the development of the ARMv1 core first began in 1983, the RISC based architecture that the revolutionary new microprocessor brought with it has been in a ongoing state of evolution. When an ARM powered embedded system is initially designed, software engineers develop system software to accompany the new platform. As the lifecycle of the platform comes to a close, the next revision of the platform is updated with a new microprocessor core and updated external periphery. This newly revised platform requires engineers to manually modify the system software originally developed for the previous release of the platform. The modifications made to the hardware of the platform must be accounted for in the new system software. Each consecutive revision made to the framework of the supporting operating system software results in the decreased efficiency of kernel and user space applications due to a non-dynamic approach to the design of the kernel's environment.

HYPOTHESIS

A dynamic kernel environment, incorporating reflective and object oriented properties, could be developed and implemented into a basic operating system. The studies conducted should entail two main improvements to kernel design theory. First, the reflective and object oriented kernel environment would allow the operating system to support any ARM based microprocessor core revision without the need to modify code to accompany an updated architecture; consequentially eliminating the need for a framework refactorization. Second, the dynamic properties of the new kernel would have an effect on the efficiency of the operating system, and decrease the overall power consumption of the microprocessor. On top of the improvements made to the kernel design theory, due to the built in type introspection of the kernel, objects could be freely linked and unlinked with the kernel during runtime.

MATERIALS

• An ARM architecture emulator. **QEMU**, a powerful open source architecture emulator that can be used to simulate the execution of ARM binaries.

• An ARM toolchain (complete with an ARM assembler, compiler, linker, and debugging tools). The **GNU ARM Toolchain**, an open source compiler and linker for the ARM architecture.

• A compatible build environment. Canonical's **Ubuntu 11.04**, an operating system environment that is compatible with the GNU ARM Toolchain.

BACKGROUND INFORMATION

The development of the microprocessor now known as the Advanced RISC Machine began in 1983 as a small team effort at Acorn Computer in response to the need for a microprocessor in the company's hardware applications. The first ARM, at the time, Acorn RISC Machine, processor, completed in 1985, became the only member of the ARMv1 series. It was developed entirely by the advanced research and development team at Acorn, a small team of developers pioneering the possibilities of embedded processing. The prototype was nothing more than an experimental version of the newfound RISC, Reduced Instruction Set Computer, based architecture. As interest in the new processor grew, Acorn set out to develop a commercial revision of their prototype ARMv1 processor. First launched the very next year in 1986, the hot off the press ARMv2 series became the first commercial RISC microprocessor. It paved the way towards the development of many applications, including the possibility of an embedded system. To this day the ARM architecture remains the infrastructure of over 90% of more than the 1 billion microprocessors found in all of the mobile devices released in 2009.

The original development team of the first revision of the microprocessor included Steve Furber, Roger Wilson, and Robert Heaton, whom are still known as the pioneers of the ARM architecture. The team's followup endeavor to the ARMv1 microprocessor was to create a chip that complemented the 32-bit RISC architecture that it introduced while maximizing the performance of the systems based around it. The team was responsible for implementing new arithmetic operations directly into the core of the processor. Real-time digital signal processing used to facilitate the creation of sounds was also implemented due to the increasing use of personal computers as a home and educational device. New compatibility features such as the addition of a co-processor interface and a low level floating point accelerator were incorporated.

Throughout the development of the processor, the team worked hard to implement the required features into a small and lightweight chip that could be easily designed and tested, as well as manufactured cheaply. One main concern was that the instruction set also needed to be small and easy to use, but at the same time powerful enough to harness the full ability of the processor. The first revision of the instruction set was presented by Wilson, who made the initial decisions to use a fixed instruction length and a load/store model following the RISC design scheme. By its release date in 1986 the ARM2 processor became the first product in the ARMv2 series. When completed it still maintained its small size and low transistor count with all of the mentioned additions. The ARM3 processor, which embodied a few modifications to help keep the series alive longer, ended the ARMv2 line of processors.

Interest in the ARM family of processors grew as more designers became interested in the RISC architecture. The ARM's design aimed to match a definite need for high-performance, low power consumption, and low-cost RISC microprocessors. In response to growing demand for their new microprocessors, Acorn split from their Acorn RISC Machine project. The processor was renamed to the Advanced RISC Machine, and thus ARM Ltd. was born. ARM Ltd. was founded with the mission to continue the development of the ARM processor and facilitate the creation of new embedded technologies. ARM Ltd's first addition to the ARM architecture was the step from the ARMv2 series of processors to the next range of microprocessors, the ARMv3 series. First introduced in 1992, the newer generation included full 32-bit addressing as well as a myriad of different features.

Since ARM Ltd’s first contribution the the scene of digital computing, it has been continuing its contribution to the development of its processor. Following the ARMv3 series came ARMv4, v5, v6, and finally ARMv7. Introduced in 2005, the ARMv7 processor series and its line of cortex processors quickly became the most widely used mobile processor core. To this day it remains the most frequently chosen processor architecture for newly developed embedded systems. The ARMv8 series of processors was announced in 2011 and has yet to be implemented into the market. It is, however, expected to be 64-bit and allow a much wider spectrum of operating system software to be run on its supported platforms.

The ARM core brought with it a new architecture using the RISC approach to computing as its infrastructure. When Acorn first announced that it needed system software developed for its new architecture, it posed a challenge to the developers of operating system software. Since the new architecture incorporated an instruction set of its own, developers had to research how the processor handled different instructions before they were able to write system software for the core. In response to the lack of support offered, a team of developers assembled by ARM collaborated to write an early version of ‘RISC OS,’ an operating system that is still being released and used on RISC based processors today. Since the development of the architecture many distributions of operating system software have been developed or adapted to work on the ARM architecture. Each operating system developed for the architecture followed the standard kernel design theory. Each operating system kernel was programmed using assembly, the C programming language, or alternate. Languages used to develop operating system software for the ARM architecture have always remained low-level, and the kernels associated with the operating systems have never run within a dynamic environment. Major contributions made to the development of operating system software for the architecture, to this day, still use the traditional non-dynamic approach to this kernel design theory.

When creating a bare-metal kernel for the architecture the conventional approach to developing the kernel's environment usually begins with the compilation of C code. C is a low-level programming language perfect for the interaction with hardware periphery and the management of core processes that is necessary on the ARM architecture. However useful C may be for the design of an operating system kernel, it lacks one feature that detracts from its usability. C works on a class to class basis, which means that when the parameters of a class need to be modified, a global variable must be set and an alternate class is responsible for making the modifications to it. Without the globalization of variables, this paradigm is rendered impossible. Classes of a typical kernel are implemented into a framework and communicate with each other using this system. A kernel that can make use of object orientation works against this principal. This is where improvements to the kernel design theory can be made. A programming language that could merge the flexibility of C and the concepts of object orientation could used to develop a dynamic kernel environment.

There are a series of programming languages that do support the implementation of object orientation. However, the presence of the flexibility of C into the programming language is crucial. After researching programming languages that meet the necessary requirements, the Objective-C programming language was chosen for three reasons. First, Objective-C is a C derivative, meaning that it retains the flexibility of C. Second, Objective-C is not an architecture specific or library reliant language, this ensures that it can be compiled for any processor architecture without the need to port any supporting libraries over with the kernel. Third, it incorporates object orientation, the key feature to the design of the proposed kernel environment. Objective-C also implements another feature that makes it ideal for the design of a low-level kernel environment; any program written in Objective-C supports reflection. The ability the programming language has to perform type introspection can be harnessed during runtime to produce yet a more efficient kernel.

The concepts of object orientation and reflection can be used to allow the class-to-class modification of objects after they have been allocated by the kernel. A kernel that is programmed using a reflective and object oriented programming language such as Objective-C has the ability to run within a dynamic environment. An operating system kernel that is supported by a frameworks composed entirely reflective and object oriented classes is able to modify the structure of each other’s parent or child objects during runtime. Merging these two properties together to form the framework of a kernel increases the versatility of the operating system as a whole, and improves the efficiency of the processor core the kernel is running on. From kernelspace, or userspace, objects requesting control of the system can link directly into the kernel during runtime, or as the system is initialized.

The paradigm of kernel processing mentioned above can be used in a virtually unlimited number of ways to carry out the procedures of a kernel. If an operating system kernel were ported from one platform to a newer platform with modified periphery, drivers for the new periphery would not have to be written and implemented into the kernel itself, a process that would require part of the kernel's framework to be rewritten in a conventional operating system. Instead, a driver object for the new peripheral devices can be developed using the kernel's standard system library and be linked into the kernel to control the interaction with the periphery. This approach to kernel design virtually eliminates the need to ever modify the framework of the kernel itself ever again in order to give an operating system control of an added peripheral device.

After reviewing many technical papers on the creation of kernel environments and making observations of the developments done to improve the kernel design theory, one thing has become clear; never before has a kernel been written in such a way that the components of its framework can interface with each other and modify various components of the system without the need for rewriting certain aspects of the kernel. This improvement made to the kernel design theory has the potential to change the way that kernel environments are designed and developed, not only for the ARM architecture, but for any processor architecture.

METHODS

The proposed approach to the design of embedded system software begins with the process of laying out the most crucial part of an operating system, its low-level firmware. The initial set of instructions known as firmware is responsible for preparing the processor to run an operating system kernel. The firmware takes the processor out of its reset state as soon as power is applied.

It then reorganizes the memory map into a known configuration accessible by the kernel. Next, it sets up the stack and all the components necessary for the execution of C and Objective-C code. Finally, it branches to the entry point of a kernel where control is essentially handed over to the framework of the operating system. (See Figure 1 for the initialization code)

Once the low-level firmware has been developed, the framework of the operating system must then be laid out. The framework of the operating system consists of the kernel and all of the classes that work with the kernel to carry out the responsibilities of system software. The development of the framework begins with the kernel setup class which is written in C. The kernel setup class is responsible for preparing to transform the active kernel environment into a reflective and object oriented environment. This is accomplished by setting any global variables required by the kernel subclass and then branching from the kernel setup (See Figure 12) to the Objective-C method responsible for initializing the primary kernel subclass. (See Figure 2 for the primary kernel subclass.)

Once the primary kernel subclass has been initialized, it sets up and administers to the interrupt controller. An interrupt controller is responsible for the handling of interrupt requests (IRQs), which allow the kernel to interrupt the normal flow of execution within the processor core to handle a specific event raised by a class. IRQs are used for a series of different exceptions, and in the case of this study, are used to to keep track of the passage of time. (See Figure 8 for the clock.) The 'IOInterrupt' framework class is used to communicate the routine and handler address of active interrupts between classes. The routine table is used to hold the handler address of the interrupts being handled by the processor. (See Figure 5 for the 'IOInterrupt' framework class and the routine table)

The ability to perform memory management is a crucial system process. A memory management unit (MMU) is implemented to allow applications to clean up after themselves during suspension or before termination, preventing unused memory from backing up the RAM and thus decreasing system efficiency. Using the MMU, programs retain the ability to allocate and deallocate the memory used by volatile variables when necessary. (See Figure 6 for the MMU)

An input/output (IO) controller is needed for the interaction between the kernel, peripheral bus, and then ultimately with a device such as an LED. The IO controller takes the provided platform bus locations from the primary ARM header (See Figure 2) and toggles the state of peripheral devices when the modification of their corresponding platform object occurs. (See Figure 5 for the IO controller, and Figure 9 for the 'OSPlatform' superclass.)

After the "boot" process of the operating system is complete, an actual application can be run. When the kernel calls an application (See Figure 10), the processor enters the proverbial "userspace." The kernel accomplishes its ultimate task by monitoring the application while administering to the rest of the system.

PROCEDURES

When transforming the theories of low-level firmware design and the techniques of reflective, object oriented kernel environment design into a physical operating system, a series of detailed procedures must be taken. This consists of following the steps of design necessary to transform an idea into an entire body of code.

First, the kernel must be based on a low-level reflective and object oriented programming language. A dedicated programming language must be chosen to write the entire kernel environment. This study focuses on the Objective-C programming language as the proprietary language due to its syntax, legibility, and its ability to induct reflectivity and create objects.

Next, an approach to programming the firmware and kernel must be decided on. This process consists of intertwining the crucial aspects of each to form a layout. The kernel's layout consists of charting the interaction between the operating system's kernel subclass (See Figure 3) and the various other parts of the kernel environment. (See Figures 2 - 11) Similarly, the layout of the firmware (See Figure 1) must be charted. When both layouts are finalized, the coding can begin.

As each class of the operating system is coded, each must be optimized for maximum efficiency. Careful attention must be paid to the syntax of the class to smooth out the kernel's flow of execution. The coding process is the simplest step in the design of an operating system. However, even though it is the simplest, it's most time consuming. This is due to insight and thought that must be spent transforming the proposed layout of the entire operating system into an idea that can be developed into code.

Once the operating system has been programmed entirely, it can be debugged. This consists of compiling and linking each of the source files of the kernel and firmware together and producing an ARM binary that QEMU is able to execute. During the execution process the activity of the processor can be monitored instruction by instruction. Once both the firmware and the kernel are working properly and bug free, testing can begin.

Each source file of the operating system is named by the compiler. When viewing the naming scheme, the flow of execution between each class must be recorded. The next procedure is choosing a target core revision. The finalized operating system binary is then executed within

QEMU. When the operating system returns, the flow of execution is logged and can be viewed. When comparing the flow of execution of the processor to the naming scheme recorded earlier, the efficiency of the kernel can be recorded by determining the number of instructions the emulator can pack throughout each cycle. The recorded efficiency of the kernel can then be compared to that of another mainstream, non-reflective and object oriented kernel such as Linux. When crunched, the numbers unveiled by the procedures above yield the findings and conclusions of the study.

FINDINGS

When comparing a kernel running within a non-dynamic environment to the operating system kernel developed using the discussed practices of reflection and object orientation to achieve a dynamic environment, the efficiency is definitely higher on the latter. When running a kernel developed in C, incorporating a non-dynamic environment, and having it perform a series of intensive processes that require the communication between members of its framework, the kernel wasn't able to complete the exchange as quickly as a kernel developed with a dynamic environment. Not only did the kernel that did incorporate a dynamic kernel environment outperform a non-dynamic kernel when it came to framework communication, but it was able to handle the manipulation standard data types quicker due to its reflective properties.

CONCLUSIONS

When the dynamic operating system kernel environment was compared to another that did not incorporate such a kernel environment, the efficiency was noticeably higher when emulating the reflective and object oriented kernel. The strain put on the processor to perform the same set of instructions was lower when emulating the dynamic kernel. These results prove that this project's theorized approach to kernel design improves the efficiency of an operating system based on a dynamic kernel. The series of procedures taken to compare the efficiency of the kernel on various microprocessor cores also confirmed that the operating system remains portable to any core revision of the ARM architecture without the need for a framework refactorization with the addition of support for modified periphery.

FUTURE APPLICATIONS

The new processes and techniques of system software development can be applied, not only to the ARM architecture, but to any processor architecture. The theorized approach to the development of a reflective and object oriented operating system kernel environment explored in this study can not only be used on a small scale, but on a large scale as well. The proposed improvement for the design of conventional kernel environments drastically improves the rate at which a kernel can handle the processing of data in its core, and in-between classes. The powerful kernel framework below each application allows the platform to process incoming data in greater quantities than ever before possible. Platforms running an operating system that is able to harness the studied approach to kernel processing could open the door to a whole new level of computing, and ultimately, to the development of scientific research currently limited by modern kernel design theory.

RECOMMENDATIONS

An experienced programmer interested in the exploring the endless possibilities of operating system development should consider reviewing this study before embarking on the design of system software for any architecture. Operating system design and development poses a manageable, yet exciting challenge to the open mind. The techniques of system software development discussed in this study can be referenced as a starting point to build system software in which there are virtually no limitations to what can be done.

ACKNOWLEDGEMENTS

I would like to thank the pioneers of embedded systems development who gave me inspiration to enter the digital race, and those who wrote the excellent books that I gathered information from. I would also like to thank my parents who make it possible to pursue what I love every day.

REFERENCES

Sloss, Andrew, Symes, Dominic, and Chris Wright. “ARM System Developers Guide” San Francisco: Morgan Kaufmann Publishers, 2004. Print

Hoal, William. “ARM Assembly Language: Fundamentals and Techniques” Boca Raton: CRC Press, 2009. Print

Furber, Steve. “ARM System-on-Chip Architecture” Boston: Addison-Wesley Professional, 2000. Print.

Knaggs, Peter, and Stephen Welsh. “ARM: Assembly Language Programming” Bournemouth, Dorest: Bournemouth University, 2004. Print.

Ryzhyk, Leonid. “The ARM Architecture” 5 June 2006. PDF file.

Ganssle, Jack, and Micael Barr. “Embedded Systems Dictionary.” San Francisco: CMP Books, 2003. Print.

Kumar, Vijay B. "Embedded Programming with the GNU Toolchain." 19 September 2011. Web. 12 February 2012. <http://bravegnu.org/gnu-eprog/>.

Samek, Miro. "Building Bare-Metal ARM Systems with GNU." *EE Times - Design*. 26 June 2007. Web. 12 February 2012. <http://www.eetimes.com/design/embedded/4007119/Building-Bare-Metal-ARM-Systems-with-GNU-Part-1--Getting-Started>.

“The ARM Instruction Set.” San Jose: ARM University Program. PDF File.

Table 1: Generic ARM Core Map

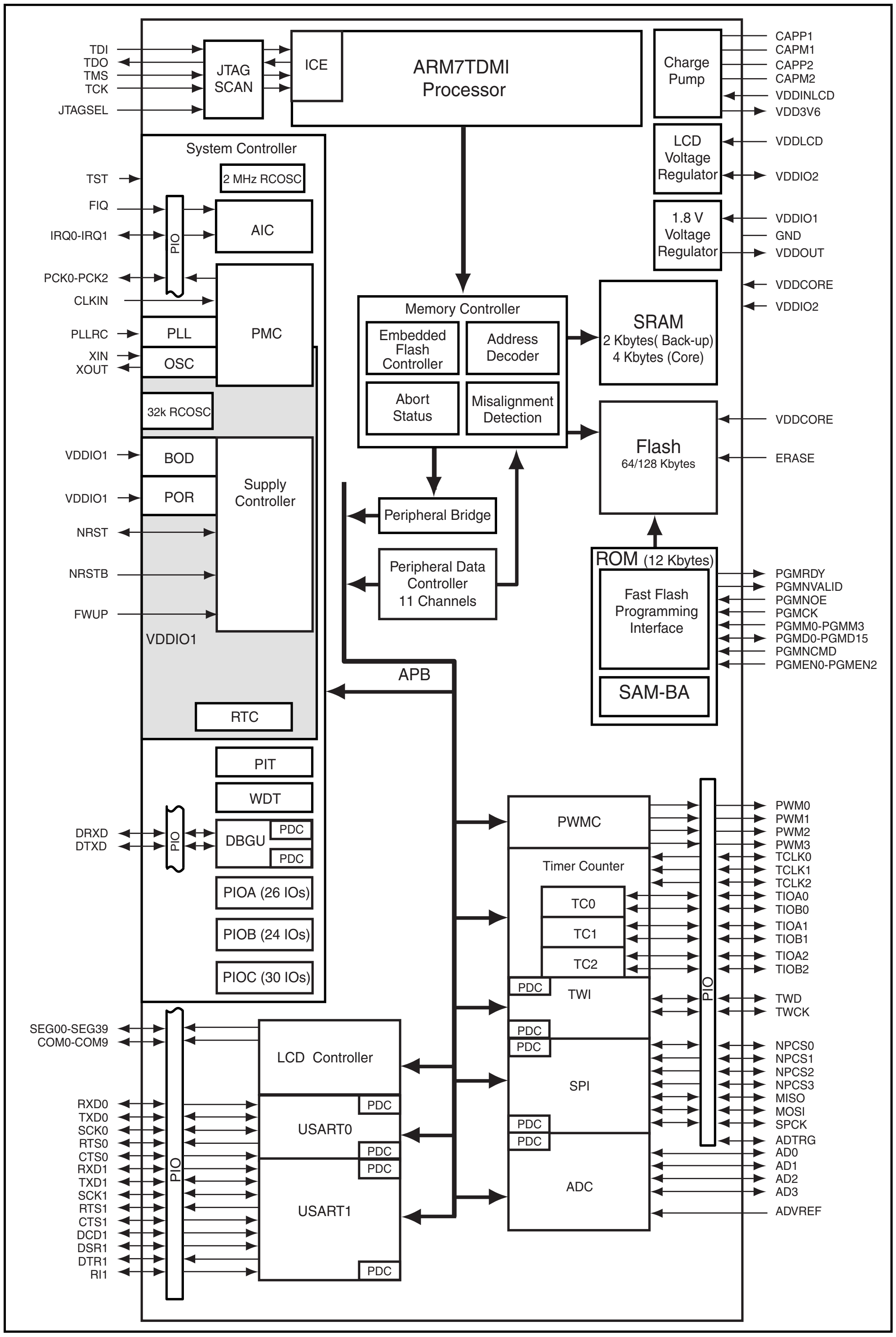
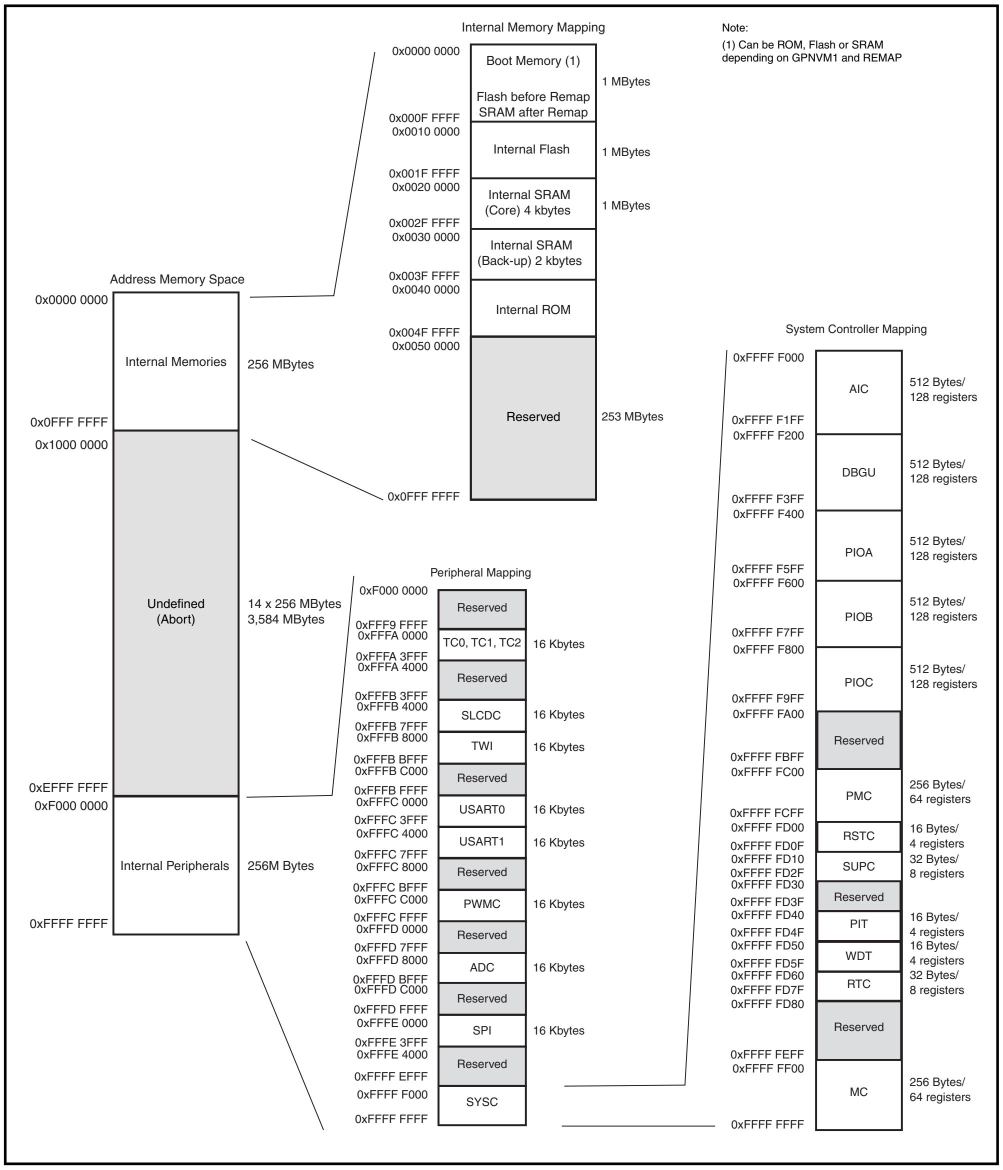


Table 2: Generic Memory Map



Atmel AT91 ARM Thumb-based Microcontroller Preliminary

Figure 1 - Firmware

```
@******************************** Interrupt Request(IRQ) Vectors ********************************@

.section "vectors" @define all of the prerequisite interrupt vectors

reset:  b  _start @tell the assembler that if the reset exception is taken, as it is on the powerup of the processor, to branch
to the address of _start

undef:  b  undef
swi:    b  swi
pabt:   b  pabt
dabt:   b  dabt

        nop

irq:    ldr pc, [pc, #0x18]

fiq:    b fiq

@************************************ Secondary IRQ Jumptable ************************************@

.global enable_irq

.type enable_irq, %function

enable_irq:
        cpsie i @enable interrupts in the the CPSR
        mov pc, lr @return

arm_irq:
        mov r13, r0
        sub r0, lr, #4
        mov lr, r1
        mrs r1, spsr
        msr cpsr_c, #(0x1F | 0x80)
        stmfd sp!, {r0, r1}
        stmfd sp!, {r2 - r3, r12, lr}
        mov r0, sp
        sub sp, sp, #(2*4)
        msr cpsr_c, #(0x12 | 0x80)
        msr cpsr_c, #(0x1F | 0x80)
        ldr r12, =irq_handler
        mov lr, pc
        bx r12
        msr cpsr_c, #(0x1F | 0x80 | 0x40)
        mov r0, sp
        add sp, sp, #(8*4)
        msr cpsr_c, #(0x12 | 0x80 | 0x40)
        mov sp, r0
        ldr r0, [sp, #(7*4)]
        msr spsr_cxsf, r0
        ldmfd sp!, {r0 - r3, r13, lr}
        nop
        ldr lr, [sp, #(6*4)]
        movs pc, lr

@******************************************** Entry Code *******************************************@

.equ STACK_POINTER, 0xA4000000

.text

_start:

        @*** move the start addresses of flash and ram into the first two registers for use ***@

        ldr r0, =flash_start @first, let r0 equal the start address of Flash in the memory map
        ldr r1, =ram_start @next, let r1 equal the start address of RAM in the memory map
        ldr r2, =data_size @finally, let r2 equal the .data segment size which was calculated in the linker script

        cmp r2, #0 @compare the .data segment's size to 0 - if it equals 0 set the 'zero' condition flag to 1

          bne _stop @halt the program if the memory map was not initialized properly

        beq init @jump to the initialization code if the memory map was set correctly

bss: @method used to initialize the BSS (Block Started by Symbol)
        ldr r0, =bss_start @first, let r0 equal the start address of the BSS
        ldr r1, =bss_end @next, let r1 equal the end address of the BSS
        ldr r2, =bss_size @finally, let r2 equal the total BSS size

        cmp r2, #0 @once again, see if r2 is equal to zero - this time it equaling the BSS size

        mov pc, lr @move the link register into the program counter to 'return' the method
```

Figure 2 - Primary ARM Header

```
//@********* This file declares the parameters for the ARM core the operating system is running on. **********@\\

#ifndef ARM
#define ARM

#define OSKernelStackPointer 0x00000000 //Define the stack pointer.

//@********************************************** CPSR Parameters *********************************************@\\

#define CPSR_N               0x80000000 //CPSR location of the N flag
#define CPSR_C               0x40000000 //CPSR location of the C flag (carry)
#define CPSR_Z               0x20000000 //CPSR location of the Z flag (zero)
#define CPSR_V               0x10000000 //CPSR location of the V flag

#define CPSR_I               0x00000080 //CPSR location of the I flag (irq)
#define CPSR_F               0x00000040 //CPSR location of the F flag (fiq)
#define CPSR_T               0x00000020 //CPSR location of the T flag (thumb)

#define CPSR_USER            0x00000010 //CPSR USR mode
#define CPSR_FIQ             0x00000011 //CPSR FIQ mode
#define CPSR_IRQ             0x00000012 //CPSR IRQ mode (the mode into which the processor will be set when interrupts are
enabled)
#define CPSR_SVC             0x00000013 //CPSR SVC mode
#define CPSR_ABORT           0x00000017 //CPSR ABT mode
#define CPSR_UNDEFINED       0x0000001B //CPSR UNDEF mode
#define CPSR_SYSTEM          0x0000001F //CPSR SYS mode

//@******************************************* Peripheral Parameters ******************************************@\\

#define platform_led_bus_location   0x00000000
#define led_power_on                0x00000000
#define led_power_off               0x00000000
#define led_color_green             0x00000000
#define led_color_orange            0x00000000
#define led_color_red               0x00000000

#endif

//@******************************************* IRQ Parameters (irq.c) *****************************************@\\

unsigned short *routines[32]; //Define the array of interrupt routines.
unsigned int clock_cycle; //Define the cycle counter.

//@******************************************* Global Memory (io.c) *******************************************@\\
```

Figure 3 - OS Kernel Superclass

```
//Import the OSObject class header.
#import "osobject.h"

//Import the OSApplication class header.
#import "osapplication.h"

#ifndef OSKernel
#define OSKernel //Define the OSKernel class header.

//@****************************************** Superclass Delagate Protocol *****************************************@\\

#pragma mark Superclass Delegate Protocol

@protocol OSKernelDelegate //Define the OSKernelDelegate's protocol and populate it.

- (void)kernelDidLoad; //Declare the delegate's primary 'kernelDidLoad' method.

@end

//@********************************************** Superclass Interface *********************************************@\\

#pragma mark Superclass Interface

@interface OSKernel : OSObject { //Declare the interface for the 'OSKernel' superclass object by basing it around the foundation
of the previously defined 'OSObject'.

    IOMemoryAddress stackAddress; //Declare the stack address variable for the superclass object.
    IOKernelStackType stackType; //Declare the stack type variable for the superclass object as well.
    OSKernelDelegate *delegate; //Declare the superclass object's delegate variable.
}

@property (nonatomic) IOMemoryAddress stackAddress; //Set the previously defined stack address variable as a property of the
'OSKernel' interface.
@property (nonatomic) IOKernelStackType stackType; //Do the same for the stack type variable.
```

Figure 4 - Primary Kernel Header and Subclass

```
//Import the OSCore framework header.
#import <oscore/oscore.h>

unsigned int *kernel_status = 0; //Declare and define an integer that can be used to moniter the kernel's current operating
state.

//@**************************************** Superclass Interface ****************************************@\\

#pragma mark Superclass Interface

@interface Kernel : OSObject <OSKernelDelegate> { //Declare the subclass' 'Kernel' interface using the 'OSObject' superclass and
OSKernelDelegate protocols.

    OSKernel *_kernel; //Preallocate the operating system's 'OSKernel' superclass object before synthisizing it to the subclass'
implementation.

}

@property (nonatomic, retain) OSKernel *kernel; //Set the operating system's previously defined 'OSKernel' superclass object as
```

```
//Import the kernel header.
#import "kernel.h"

//Import the memory read/write functions from the IO controller.
#import <arm/io.h>

//@*************************************** Subclass Implementation ***************************************@\\

#pragma mark Subclass Implementation

@implementation Kernel //Implement the 'Kernel' interface.

@synthesize kernel = _kernel; //Synthesize the operating system's 'OSKernel' superclass object to the 'Kernel' implementation,
sealing the object as the interface's property while linking it to the preallocated 'OSKernel' superclass object.

- (id) init:(id)object override:(id)override { //Populate the initialization method's declaration for the 'Kernel' interface's
implementation.

    if (self = [super init]) { //If the implementation's super 'OSObject' initialization method returned equal to the
implementation's self, return.

        OSKernel *systemKernel = [OSKernel init]; //Create a nil 'OSKernel' object that will become the kernel of our operating
system.

        [systemKernel allocateStackWithAddress:OSKernelStackPointer stackType:IOStackTypeAscending]; //Set the operating
system's primary stack pointer

address.

        [self setKernel:systemKernel]; //Set the system's kernel object as the defined 'OSKernel' instance.

        [systemKernel validate]; //Validate the kernel object and officially enter kernel-space.

        return (override) ? override : self; //Return the override for the parent 'OSKernel' object if it exists, otherwise,
default to returning self.
    }

    return object; //Return the specified initialization object to the implementation.
}

- (void) branch {
    [[IOMemoryController memoryController] initialize]; //Initialize the operating system's memory controller.

    [self init:nil override:nil]; //Initialize the kernel subclass with no specific platform override properties.

    return; //Jump back to the kernel's main C entry function if the kernel were to return.
}

//@******************************************** Delegation Methods ********************************************@\\
```

Figure 5 - OS Interrupt Superclass and Interrupt Controller

```
//Import the OSObject class header.
#import "osobject.h"

#ifndef OSInterrupt
#define OSInterrupt

//@********************************************** Superclass Interface *********************************************@\\

#pragma mark Superclass Interface

@interface OSInterrupt : OSObject {
    int routine;
    unsigned short *handler;
}

@property (nonatomic) int routine;
@property (nonatomic) unsigned short *handler;

- (void)interruptWithRoutine:(int)routineIndex handler:(void *)handlerAddress;

@end

//@*************************************** Superclass Implementation **************************************@\\

#pragma mark Superclass Implementation

@implementation OSInterrupt

@synthesize routine, handler;

- (void)interruptWithRoutine:(int)routineIndex handler:(void *)handlerAddress {
    [self setRoutine:routineIndex];
    [self setHandler:handlerAddress];
}

@end

#endif
```

```
//Import the primary ARM header.
#import "arm.h"

unsigned short *routines[32] = { //Define 32 address holders for external ISRs (Interrupt Service Routines).
    0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000
};

void irq_add_handler(int routine, unsigned short *address) {
    routines[routine] = address; //Add the address of the ISR to the routine table.
}

void irq_remove_handler(int routine) {
    routines[routine] = (unsigned short *)0x00000000; //Remove the address of the ISR to the routine table.
}

void init_irq() {

    //*** Enable IRQ (Interrupt ReQuests) in the CPSR (Current Program Status Register) ***\\

    __asm("cpsie i");
    __asm("mov pc, lr");

}
```

Figure 6 - Memory Management Unit (MMU)

```
//Import the OSCore framework header.
#import <oscore/oscore.h>

#ifndef IO
#define IO

//@********************************************* Superclass Interface *********************************************@\\

#pragma mark Superclass Interface

@interface IOMemoryController : OSObject {

}

+ (IOMemoryController *) memoryController; //Create a singleton instance as the 'IOMemoryController' interface's shared handler;

- (void) initialize;

- (unsigned short *) writeValue:(unsigned short *)value toAddress:(IOMemoryAddress)address;
- (void) copyValueAtAddress:(IOMemoryAddress)fromAddress toAddress:(IOMemoryAddress)toAddress offset:(int)offset;
```

```
//Import the IO header.
#import <arm/io.h>

@implementation IOMemoryController

static IOMemoryController *memoryControllerHandler = nil;

+ (IOMemoryController *) memoryController {

    if (memoryControllerHandler == nil) {

        //Any variables needed to set up the memory controller will be defined here.

    }

    return memoryControllerHandler;
}

- (void) initialize {
    //Normally, the next step here would be to set up the hardware memory controller, which allows the operating system to
interact with memory peripherals, however, since we are emulating the operating system, the initialization of this controller is
not needed.

    /* Hardware Memory Controller Initialization Not Required Within Emulation */

    //Next, initialize the operating system's memory controller by copying the .text and .data sections from flash memory into
RAM.
    __asm ("ldrb  r4, [r0], #1");
    __asm ("strb  r4, [r1], #1");
    __asm ("subs  r2, r2, #1");
    __asm ("bne  copy");
}

- (unsigned short *) writeValue:(unsigned short *)value toAddress:(IOMemoryAddress)address {
    unsigned short *destination = address;

    *destination = value;

    return value;
}

- (void) copyValueAtAddress:(IOMemoryAddress)fromAddress toAddress:(IOMemoryAddress)toAddress offset:(int)offset {
    const char *source = (const char *)fromAddress; //Load the value of the given source address.
    char *destination = (char *)toAddress; //Create a pointer to the destination address.

    for (; increment != 0; increment --)
        *destination ++ = *source ++;
```

Figure 7 - OS Object Protocol Superclass

```
//Import the primary ARM header.
#import <arm/arm.h>

#ifndef OSObject
#define OSObject //Define the OSObject class header.

typedef int BOOL; //Define boolean and its values for use within the Objective-C.

#define FALSE 0
#define TRUE  1

#define nil (id)0 //Define the 'nil' Objective-C object as a blank 'id.'

//@***************************************** Superclass Protocol *****************************************@\\

#pragma mark Superclass Protocol

@protocol OSObjectProtocol //Define the 'OSObject' superclass protocol and populate it.

typedef unsigned short *IOMemoryAddress; //Typedef an unsigned short pointer as an IOMemoryAddress object.

typedef enum { //Create an enumerator to hold each of the different stack types.
    IOStackTypeDefault, //Create an standard enumerator varible for the processor's default stack type. (Ascending)
    IOStackTypeAscending,
    IOStackTypeDescending,
} IOKernelStackType; //Seal the enumerator as an IOStackType object.

typedef enum { //Create an enumerator to hold each of the different processor registers.
    IORegister01,
    IORegister02,
    IORegister03,
    IORegister04,
    IORegister05,
    IORegister06,
    IORegister07,
    IORegister08,
    IORegister09,
    IORegisterR10,
    IORegisterR11,
    IORegisterR12,
    IORegisterSP,
    IORegisterLR,
    IORegisterCPSR
} IORegister; //Seal the enumerator as an IORegister object.

- (id) self; //Declare the 'self' object for use within subclasses.

@end

//@***************************************** Superclass Interface ****************************************@\\
```

Figure 8 - Clock

```
//Import the primary arm header.
#include "arm.h"

void delay(int time) { //In milliseconds.
    unsigned long tick = clock_cycle + time;

    while(clock_cycle < tick);
}

void clock_second() {
    //Every second, this function will be called.

    __asm ("add r4, r4, #1"); //Register 4 will act as a system timer.
}

void clock_handler() {
    clock_cycle ++; //Every CPU cycle, the cycle counter is incremented by 1.

    if (clock_cycle % 18 == 0) //Every 18 cycles, 1 second will have passed.
        clock_second(); //Call the 'second' function.
}
```

Figure 9 - OS Platform Superclass

```
//Import the OSObject class header.
#import "osobject.h"

#ifndef OSPlatform
#define OSPlatform

typedef enum { //Create an enumerator to hold the different colors that the platform's LED can support.
    OSPlatformLEDColorGreen,
    OSPlatformLEDColorOrange,
    OSPlatformLEDColorRed
} OSPlatformLEDColor; //Seal the enumerator as an OSPlatformLEDColor object.

typedef struct { //Create a struct to hold the state of the LED, its ON value, and color.
    BOOL on;
    OSPlatformLEDColor color;
} OSPlatformLED; //Seal the struct as an OSPlatformLED object.

//@********************************************** Superclass Interface *********************************************@\\

#pragma mark Superclass Interface

@interface OSPlatform : OSObject {
    OSPlatformLED _led;
}

@property (nonatomic) OSPlatformLED led;

+ (OSPlatform *) currentPlatform;

@end

//@************************************** Superclass Implementation *************************************@\\

#pragma mark Superclass Implementation

@implementation OSPlatform

@synthesize led = _led;

static OSPlatform *platformHandler = nil;

+ (OSPlatform *) currentPlatform {
```

Figure 10 - 'Blink Application'

```
//Import the primary operating system framework header.
#import <oscore/application.h>

#pragma mark Application Interface

@interface Blink : OSApplication <OSApplicationDelegate> {
    OSPlatformLED led;
}
```

```
#import "Blink.h"

@implementation Blink

#pragma mark OSApplication Delegate

- (void) applicationDidFinishAllocating {
    led = [[OSPlatform currentPlatform] led];

    [led setOn:TRUE];

    for (i = 100; i != 0; i --) { //Do this 100 times.
        [led setColor:OSPlatformLEDColorGreen];
        delay(1000); //Delay for 1 second (1000 milliseconds).
        [led setColor:OSPlatformLEDColorOrange];
        delay(1000); //Delay for 1 second (1000 milliseconds).
        [led setColor:OSPlatformLEDColorRed];
        delay(1000); //Delay for 1 second (1000 milliseconds).
    }

    [led setOn:FALSE];

    [self kill];
}
```

Figure 11 - OS Application Superclass

```
//Import the OSObject class header.
#import "osobject.h"

#ifndef OSApplication
#define OSApplication

typedef enum { //Create an enumerator to hold each of the different application types.
    OSApplicationTypeSystemApplication,
    OSApplicationTypeUserApplication
} OSApplicationType;

//@***************************************** Superclass Delagate Protocol ****************************************@\\

#pragma mark Superclass Delegate Protocol

@protocol OSApplicationDelegate

- (void) applicationDidFinishAllocating;
- (void) dealloc;

@end

//@********************************************* Superclass Interface ********************************************@\\

#pragma mark Superclass Interface

@interface OSApplication : OSObject {

    IOMemoryAddress address; //Declare a variable that will store the application's memory address.

}

- (void) alloc;
- (void) launch;
- (void) suspend;
- (void) kill;

@end

@implementation OSApplication

- (void) alloc {
    //Allocate the application in memory and call the delegate.
    [[self delegate] applicationDidFinishAllocating];
}

- (void) suspend {
```

Figure 12 - Kernel Setup

```
//Import the kernel header.
#include "kernel.h"

int kernel() {

    //Set the system up for the execution of Objective-C code.

    branch(); //Jump to the 'branch' method within the Objective-C class "kernel.m".

    return &kernel_status; //Return the current system status of the kernel, jumping back to the entry code during shutdown.
```