Operating System Design and Development Continued

George Morgan

March 2013

Year 2

ABSTRACT

The release of UNIX in 1969 set the standard for operating system design and development. Over the past 40 years, programming language theory has seen the inception of paradigms, novel models of structural organization, such as object orientation and polymorphism, none of which UNIX take advantage of. Since then, nothing significant has been done to keep UNIX up to speed with the ongoing evolution of programming languages. The problem is that most modern day applications are written in paradigmatic programming languages, but the system software, like UNIX, that is responsible for managing their resources is still being written in a non-paradigmatic language.

The objective of this project was to apply modern paradigmatic programming to in-depth operating system design and development. What was found was that the models of locality of reference and differential inheritance, along with modern programming paradigms such as object orientation and polymorphism, had the potential to enable a given operating system to handle jobs and manage the resources of high-level applications more efficient due to the elimination of common inefficiencies such as duplication of code in memory, redundant instruction execution, and incoherent code structure. Tests confirmed that, by applying the principles of paradigmatic programming to the operating system kernel developed in last year's study, the resulting operating system was capable of operating at a higher level of abstraction than any other UNIX-like non-paradigmatic operating system has for over 40 years.

STATEMENT OF THE PROBLEM

Since the release of UNIX in 1969, nothing significant has been done to refine the way that operating systems are being designed. High-level programming languages brought with them a multitude of revolutionary new and abstract concepts such as object-orientation and reflection, both of which were

explored in last year's study. But operating systems aren't where innovation is being done and the same design principal for operating system software has been reused for over 40 years.

HYPOTHESIS

By applying the principals of modern programming language theory to the framework of an operating system, along with the same founding principals of last year's project, a better refined operating system kernel can be developed that will set the standards for modern operating system design and development while simultaneously bringing the same profound increase in efficiency that was seen in last year's research.

MATERIALS

- A target architecture. The **ARM** architecture.
- An target ARM platform. The **AT91SAM7S321**. (See **Figure 4** for block diagram.)
- A build environment. **Mac OS X Mountain Lion**.
- An integrated development environment. **Xcode**.
- A package manager for Mac OS X. **Homebrew and MacPorts**.
- An ARM toolchain. The **GNU ARM Toolchain** ported to Mac OS X.
- An application performance analyzer. **Instruments**.

BACKGROUND INFORMATION

In 1969, a small project headed by Ken Thompson and Dennis Ritchie, which was later named UNIX, marked the beginning of the standardization for the design and development of operating system software. The project was originally aimed to refine an earlier, experimental operating system named Multics; but the

research done ended up leading to much more than just a refined operating system. Thompson and Ritchie worked to develop new concepts that would improve the efficiency, portability, and the ease of use of **UNIX**. Some of the new concepts that were introduced include: the hierarchical file system, individual processes, device files, a command line interpreter, and most importantly, the implementation of the C programming language.

The development of UNIX was preceded by many advancements in programming language theory. Before Thompson and Ritchie began work on the operating system, a lot happened behind the scenes of language design. The dawn of computing brought with it an interest in making programming languages more accessible to users that weren't familiar with or interested in learning the intricate and convoluted operating codes that computers were programmed in at the time. The assembly language was the first programming language that succeeded at making programs easier to read and write, but it was still confusing and required knowledge of low-level commands. There was still a long way to go before engineers reached their goal of making a language look like english. The first programming language to remotely accomplish this goal was Fortran. What the developers of Fortran succeeded in doing was raising the level of abstraction between the program and the hardware of the computer. Things like the computer's memory addresses, registers, and operation codes were hidden behind a mask, a more readable syntax. Programmers found was that they could generally be much more productive using Fortran because of its readability. Fortran's success set a standard for the design of future programming languages. It introduced many of the keywords that are still found in programming languages today. If, do, continue, return, and end are all found in the Fortran syntax. Fortran also introduced the function.

Following the development of Fortran came a group of languages that can all be classified in the same category. COBOL was the first experimental language that aimed to reach the goal of making programming languages look more like english after the attempt made by Fortran. It wasn't much of a

success, but it was a major stepping stone in the development of the programming language theory and helped future language designers learn from its mistakes. BASIC was developed by stripping Fortran down to its simplest possible parts and was the next language to take over where COBOL left off. It was quick to learn, and it quickly became popular. Microsoft BASIC, which later became Visual BASIC, was introduced by Microsoft shortly thereafter. BASIC played a huge role in establishing Microsoft as a company and served as the foundation for Microsoft's Windows Operating System. ALGOL 60 was the biggest development in programming language theory since the introduction of Fortran. It introduced the notion of structured programming and blocks. It was an extremely important and influential language due to the concepts it introduced. Structured programming debates arose from its introduction, and it helped lead to the abolishment of use of the "goto" operation.

Each of the languages developed during the zenith of programming language design were very specialized. There was interest in developing a common language that could be used in a generic context. However, despite the many attempts that were made to make this conceptual language, there wasn't any success. In fact, there was a reaction to these attempts that suggested that languages should be simplified. These suggestions were taken into account and ALGOL 60 evolved into Pascal, a simplified general purpose programming language. It had a number of major design flaws that came with it, it wasn't modular enough, and its system was flawed, but it opened the door towards the development of the most influential programming language of all.

All throughout the early years, there was an enormous architectural variety between computer systems. Each architecture varied on basics such as: word size, number types, register count, register purpose, and instruction sets. The development of UNIX inspired its designers to create a new programming language that would enable their operating system to be ported to any platform or architecture that had supporting compiler. This concept was game changing because once an operating system was written for

one architecture, it was inherent that it be rewritten using an entirely different instruction set in order to port it to an alternate architecture. The ongoing evolution of programming languages inspired Ken Thompson, co-writer of the UNIX operating system, to write a language which he called B. It took the strengths of BCPL, one of the successors of ALGOL 60, and PL/I, a successor of COBOL, and gave them the syntax of Fortran. Although Thompson's attempt to develop a programming language was noble, B didn't serve as much of an improvement of what the world had already seen. The scene of programming language design changed for good when Dennis Ritchie took Thompson's B, and created C, a programming language which combined the type system of Pascal with the syntax and framework of B. UNIX, which was originally written in assembly language, was rewritten in 1972 in Ritchie's C programming language. Ritchie's implementation of an operating system using a language that existed at a higher-level than assembly defied the popular notion "that something as complex as an operating system, which must deal with time-critical events, had to be written exclusively in assembly language." [1] However, Ritchie proved this notion false when the publication of the newly written UNIX operating system marked the first major development in the history of computer science since the inception of the idea of an operating system itself. Along with the development of C came many breakthroughs to the design of programming languages. C became incredibly popular and has since become the most important implementation language of all time. Virtually every languages since its development has either been implemented in C or is based on C. The language brought with it the formalities for structured programming, lexical variable scoping, recursion, and a static typing system.

As time went on, computer engineering no longer began to see radical changes being made to architecture design. "Nobody's making new stuff. Nothing radical. It's just refinements of stuff that's been happening for several decades." [2] Today, there are only three primary architectures that are recognized by the standards of software design. Those select architectures are the Intel architecture, the Power PC

architecture, and ARM architecture. "We're doing even worse in Operating Systems. It used to be that every model of every machine had its own Operating System. And that came with a lot of obvious inefficiencies. But there's no innovation happening in operating systems. Basically, we've been rewriting the same systems for 40 years. It's just not where we do innovation. Where we do innovation is in programming languages. And that's been going on for quite a long time." [3]

We now only have two operating systems: UNIX and Windows. (See **Figure 1** for a tree of UNIX based operating systems.) There is the debate, which level of languages should we use to write applications for these operating systems? High-level language increased productivity, but low level languages saved space and overhead. It was decided on that high-level languages be used for implementation of software, the decision that led to the development of Simula, the first object-oriented programming language. Simula was highly influential on Smalltalk, the first modern object-oriented language that implemented the paradigms of object-orientation that can be seen in all object-oriented languages today. Smalltalk brought with it early terminology, i.e. “sending a message” is now what we call “invoking a method.” Smalltalk pushed the evolution of C to the next level, and ultimately, Objective-C was born. Objective-C is an object-oriented language that was created primarily by Brad Cox and Tom Love in the early 1980s. It added Smalltalk style messaging to the C programming language, and was used as the foundation for the development of the operating system introduced in last year’s project.

METHODS

One of the primary goals of this year’s study was to take the concepts introduced in last year’s research and make them architecture inspecific. The ARM architecture is still being used as the target architecture in this project, but the research that has been done is applicable to any processor architecture such as: the ARM architecture, the AVR architecture, the PIC architecture, the PowerPC architecture, the

x86 architecture, and many more. In order to extend functionality of a platform inspecific operating system to a specific architecture, a method of linking what is referred to in this project as architecture support bundles and platform support bundles with a dynamic library containing the source of the operating system was devised so that the operating system can be distributed to developers so that support for third-party platforms can be extended to the operating system without the need for releasing its source. The support bundles contain preprocessor definitions, entry code, and specialized functions specific to the target platform or architecture. Both the architecture support bundle and the platform support bundle were programmed to extend support for the target platform, the SAM7S, and architecture, the ARM architecture, to the operating system.

A different, better refined approach to operating system kernel design and development was taken in this year's study. The concepts of object-orientation and reflection that were introduced in the last year's study are still an integral part of the framework of the improved kernel environment, but a major step forward was taken in an effort to gain finer control of the properties of Objective-C, the programming language used in the previous year's research. Object-Oriented languages are reliant on a runtime library. The runtime of an object-oriented programming language is responsible for extending the properties of object-orientation to a language that doesn't support it. Objective-C is an extension of the C programming language. It uses its runtime to extend support for its properties to C. Due to the nature of Objective-C, an Objective-C compiler simply takes the syntax of Objective-C and translates it into C, replacing Objective-C method calls with a function call to the runtime library responsible for sending a message to an object in expectation of retrieving a value returned from a function inside the structure of a class object. Extended properties of object-orientation, such as reflection, are hard wired directly into the runtime library using C. Due to this phenomenon, there is a medium that is reached in compiler-space when Objective-C code is translated entirely into C code, which is subsequently or parallelly compiled using a standard C compiler.

This means that each and every time a program is compiled, the Objective-C runtime library has to be linked against the binary of an application written in Objective-C. What this means in terms of operating system design and development is that parts of the kernel can be written in both C and Objective-C and be equally efficient if the runtime library of the programming language is implemented directly into the framework of the operating system kernel instead of being linked against it.

What was found by doing last year's study was that when the operating system was compiled, the Objective-C runtime was called every time the operating system messaged an object, including when user-level applications made system calls. Although the original runtime library was efficient to begin with, what was found was that the operating system kernel itself did not have control of what happened within it because it was linked in as a library rather than directly implemented in the source code. This was met with a number of inefficiencies. First, there were extensions to the Objective-C runtime that were not pertinent to the range of functionality of the operating system, which caused bloating of the final kernel binary. Second, extraneous calls were being made to parts of the operating system that were not yet supported by the kernel. The only way to combat these issues was to develop a custom runtime environment and implement it directly into the operating system's framework so that not only could user-level applications access the runtime from user-space, but so could objects from kernel-space. The observation of the behavior of the runtime library when implemented directly into the framework of the operating system led to the conclusion that the overall efficiency of the kernel was improved. This observation can be explained simply as the combination of two principles: the principle of locality of reference, which states that if a particular function or memory address is accessed once, it is likely to be accessed again, and the law of differential inheritance, a model that operates on the principal that most objects in object-oriented programming, either class or prototype based, are derived from other more generic objects. These two principles can be used to further refine the design and development of the operating system by decreasing code density, the

eliminating duplication of functions in memory, and decreasing the segregation between user level applications and the operating system kernel, all of which have the potential to dramatically change the course of the operating system design theory.

The locality principal, also referred to as locality of reference, as discussed in the paragraph above is one of the predictable behaviors of a computer. Temporal locality, a subdivision of the locality principal, states that if a particular function or memory address is accessed once, it is likely to be accessed again during runtime. Spacial locality, another subdivision of locality of reference, states that, like temporal locality, if a particular function or memory address is accessed once, surrounding memory addresses and functions will be accessed again in the near future. A phenomenon, which, for the sake of this study, is referred to as acceleration was developed by fusing the principals of temporal and spacial locality to combat the inefficiencies of the traditional method of iterative computation, which, also for the sake of this study, will be referred to as procrastination. In a conventional method of iterative computation, take for example, when the integrity of a function is verified, the operating system kernel has to interrupt the task that it is currently doing, check the integrity of the function, and then resume accordingly. The process of procrastination takes a fixed number of instructions per iteration and is relatively consistent across the kernel environment. (See **Figure 1** and **Figure 2** for a graph of the projected instructions per iteration of procrastination.) By using objects as kernel entities, each has a cache in memory that stores its specific properties. If a message request is made, the kernel can verify its integrity and write to the method cache so that if it is requested again in the future, the kernel doesn't have to spend the time performing the integrity check. (See **Figure 1** and **Figure 2** for a graph of the projected instructions per iteration of acceleration.) Although acceleration requires a number of extra instructions to initially write a value to the cache, it is much more effective to do so in the long run in justification by locality of reference. This particular

paradigm can be applied to numerous other aspects of operating system design and can lead to an increase in overall efficiency due to minimization of redundancy and repetition of instructions.

The notion of breaking away from a class specific interface and implementing the kernel using a hybrid interface which consists of both classical and prototypal interfaces was considered for incorporation in last year's project, but never made it into the research because last year's project mainly focused on the newfound concept of implementing an entire operating system kernel using a class based, object-oriented programming language. Most object-oriented languages are based on a classical framework. This means that each class is actually an object located in memory that contains variables and methods, and may or may not be based on a superclass. The antithesis of a classical framework is a prototypal interface, a concept first introduced to the programming theory with the Self programming language. In a prototypal environment, classes are not present. Objects can be created, destroyed, and modified during the runtime of an application without any overhead other than a runtime library. What was game changing about the layout of the framework of last year's study was that by incorporating the concepts of object-orientation and reflection, classes had the ability to communicate interactively. However, all classes were objects themselves. Classes and their objects do not have the ability to be created and destroyed. This is not a problem when extending functionality of the operating system kernel by linking ulterior class objects such as drivers or kernel units into the framework of the kernel. However, once these classes were added, they could not be modified, created or removed. In some cases, this is ideal. Fixed kernel units such as the memory management unit or the interrupt controller are implemented as classes because they are destined to be laid out in memory and never destroyed. However more modular parts of the kernel such as the resources administered to user level applications need to be dynamic enough to be created, destroyed, and structurally modified during runtime. The obvious solution to this proposition is a strictly modified prototype based

interface that can be implemented alongside the traditional classical framework introduced in last year's project.

Instead of recreating superclass objects in memory when defining subclasses, which is what is traditionally done in object oriented programming, the refined approach towards operating system design and development follows the model of differential inheritance, a model that is based on the principal that the objects of one prototype or class are likely to share their properties with more generalized objects. This approach can be viewed in direct relation to polymorphism, particularly parametric polymorphism because each data type can be instantiated generically. For instance, all objects regardless of their type share the general feature of allocation. Instead of recreating a given function such as *allocate()* in memory, making a function pointer to its address, and distributing it throughout the object implementation as it is called, a master object is created in memory. Whenever an object requests the creation of a variable, the prototype of the object can modularly point directly to the generalized object, which results in a clean but intricate class network with absolutely zero memory duplication.

Reusability, in terms of computer science and object-oriented programming is synonymous with differential inheritance. The simplest form of reuse consists of subroutines, or functions being shared between two class objects. In computer science, polymorphism, as mentioned earlier, can simply be defined as a feature of an object-oriented programming language that allows different data types to be handled using a uniform interface. The concepts of parametric polymorphism apply to both data types and functions. A polymorphic object, i.e. a polymorphic function can evaluate, or be applied to values of different types. Ad hoc polymorphism is a kind of polymorphism in which polymorphic functions can be applied to arguments of different types. A polymorphic data type is a generalized data type from which specializations are made, which is a key concept of reusability. When objects are reused, such polymorphic objects serve as the foundation for more specialized class objects.

PROCEDURES

The first step taken to kick off this year's project was bringing the product of last year's research back to the drawing board. All of the aspects of the operating system specific to the ARM architecture were stripped out and set aside making the kernel architecture inspecific. Because support for the ARM architecture was no longer built into the operating system directly, it had to be plugged back in. Instead of following the convention of compiling the operating system along with support for its target architecture, a new method had to be devised. This led to the next step of the project, developing support bundles. The first of two support bundles, the architecture support bundle, consisted of the headers and supporting files stripped from the last year's project that extended functionality to ARM architecture. These files were simply recompiled and linked against a dynamic library consisting of the operating system's source code to create a binary specific to the target architecture. The second support bundle, the platform support bundle, contains information about the target platform, which had yet to be decided on.

In order to continue with the research, a target platform had to be decided on. After searching for hours on end, Atmel's AT91SAM7S321 chipset, or SAM7S for short, was selected. (Refer to **Figure 4** for platform specifications of the SAM7S chipset.) It was powerful enough to run the tests that were scheduled for this year's research; clocking in at at a maximum operating frequency of 55 MHz, it has just enough processing power to run an operating system based on an object-oriented framework. The 321 model of the SAM7S is identical to the 32 model but has a built in USB controller, a feature that would simplify the programming and debugging the processor. The structure of the chipset wasn't overcomplicated and has just enough memory to store the operating system and high-level applications. Perhaps most importantly, the SAM7S is an ARM7TDMI based processor, the simplest and most common revision of the ARM

architecture. Once a supporting development board was purchased, the platform support bundle could be coded and optimized for object-orientation, restoring functionality to the operating system.

Next, it was simply a matter of doing research, drawing out concepts, and performing simple tests. This process took the majority of the time spent on this project. The first thing that was done was centered mainly on the framework of last year's operating system. One of the flaws of last year's project was that it failed to directly pinpoint the part of operating system that had such a dramatic effect on its efficiency. It was known that object-orientation was the cause of the improvement, but it was unknown what part of the language was responsible. The first thing that this year's research set out to explain this missing link. After a series of tests that revealed that the runtime library of the Objective-C programming language was responsible, it was clear where the development of this year's project would be taken. (See **Methods** for a detailed explanation of why the runtime library was responsible for the increase in the efficiency of the kernel environment.)

To further refine last year's operating system, the runtime was extracted from the base of the Objective-C programming language and rewritten to shape a runtime that could be implemented directly into the framework of the operating system. The implementation of the language runtime directly into the framework of the operating system opened the doors to possibilities that weren't before possible when the runtime wasn't accessible by the operating system kernel. Being able to access the runtime directly from the kernel environment allowed messages to be overridden, redirected, and requested at any given time. This means that the operating system could directly control the cache, implementations, and other features of a runtime library to administrate not only to the object-orientation of the operating system, but to its framework as well.

The fine control that the operating system has over the runtime due to its implementation directly into the framework of the operating system itself led the way into new territory of operating system design

and development. The way that each object could is handled in memory could be used to refine the conventional methods of iterative computation. A phenomenon which, for the sake of this study, is referred to as acceleration was developed to combat the inefficiencies of procrastination. (See **Methods** for a detailed explanation of procrastination and acceleration and **Figure 1** and **Figure 2** for a graph comparing the efficiency in instructions per iteration of the two.)

Once a foundation for the refinement of the design of last year's operating system kernel was laid out, further improvements could be made. Numerous paradigms of modern programming language theory were implemented directly into the framework of the operating system. First, polymorphism, as applied specifically to operating system design and development, enabled certain kernel class objects or prototypes to act as superclasses for more specialized class objects without duplicating object mutualities in memory. For example, according the the principal of differential inheritance, a generic class that consists of a universal set of variables and a set of static functions, such as *allocate()* and *dealloc()***,** can be the superclass of a further specialized class object. Instead of duplicating the superclass object in memory, as is traditionally done in object-oriented programming, the subclass can simply redirect messages sent to its static functions to the superclass' functions in memory, reducing both the size of the object and the size of the kernel in memory.

All of these features which are scheduled to be implemented directly into the framework of the operating system require the framework of the operating system to be refactored completely. This is the largest and most time consuming procedure of the project, developing the operating system itself using the above principals of kernel design. The way that these programming paradigms are implemented into the framework of the operating system is by carrying out the above procedures and finalizing the operating system by optimizing it for performance.

FINDINGS

The first profound development that was made in this year's research was the explanation of why an object-oriented and reflective framework had such a dramatic effect on the efficiency of the last year's operating system. Tests on each individual part of last year's operating system framework eventually led to the conclusion that the runtime of the Objective-C programming language, specifically its ability to perform message sending, was responsible for this phenomenon. The subsequent developments aimed to further refine last year's operating system. It was discovered that that the reorganization of the operating system by implementing the runtime environment directly into the framework enabled user-space and kernel-space class objects to make use of language paradigms not made possible when statically linking the runtime against the operating system binary. As a result, new concepts such as extensive polymorphism and prototypal interfacing could used by the kernel, both of which had a dramatic impact on the kernel's ability to manage its resources effectively. Polymorphism, as applied to operating system design and development, improved the management of resources within the kernel due to the reduction of code duplication in memory. Acceleration, a proposition aimed to minimize the redundancy of iterative computation proved to have a dramatic effect on the elimination of repeated instructions when performing iterative processes. Structured programming, in terms of the frequency of use of libraries, reduced the amount of unused code taking up space in memory. The application of the locality principal towards members of classes and prototypes ensures that the overhead of some runtime specifics would be justified. Objects that needed to stay static in memory could be implemented as classes. Whereas objects that needed to be dynamic could be defined as prototypes. Overall, the implementation of the programming language theory and the paradigms associated with each into the kernel environment had a dramatic impact on the way that operating system functioned. As a result, the operating system handled messaging more dynamically, retained a linear approach to class networking, and managed its resources more efficiently.

CONCLUSION

In conclusion, the application of modern programming theory to the design and development of an operating system framework, on top of the concepts of object-orientation and reflection which were introduced in last year's study, resulted in a dynamic kernel environment capable of operating at a higher level of abstraction than any other operating system has for over 40 years. As a result of the research and development done in this study, user-space applications as well as kernel-space processes running within the refined kernel environment were given the ability to harness programming paradigms without any overhead of their own. Not only were the features implemented directly into the operating system's framework applicable to the organization of high-level and low-level procedures, but also to the improvement of the overall stability and efficiency of the operating system due to phenomenon such as: acceleration, polymorphism, locality of reference, structured programming, prototypal interfacing, and more. The improvements proposed to refine the operating system design theory have the potential to set the standard for how modern operating systems are designed and developed in the future.

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.

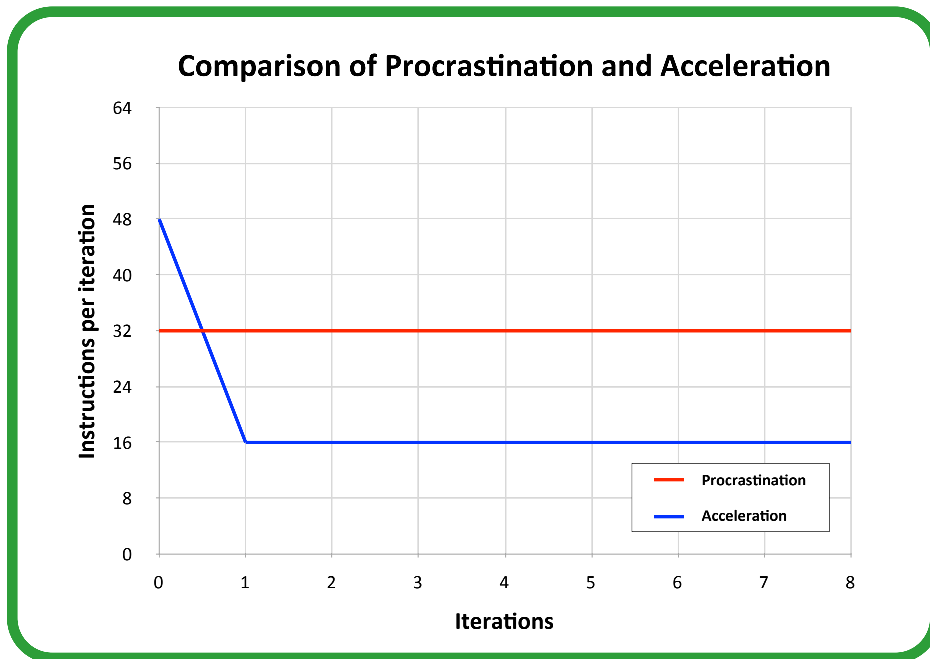

Figure 1: Acceleration, represented by the blue line, takes a number of extra instructions to initialize but is much more effective in the long run than procrastination, the conventional method of iterative computation, which is represented by the red line.

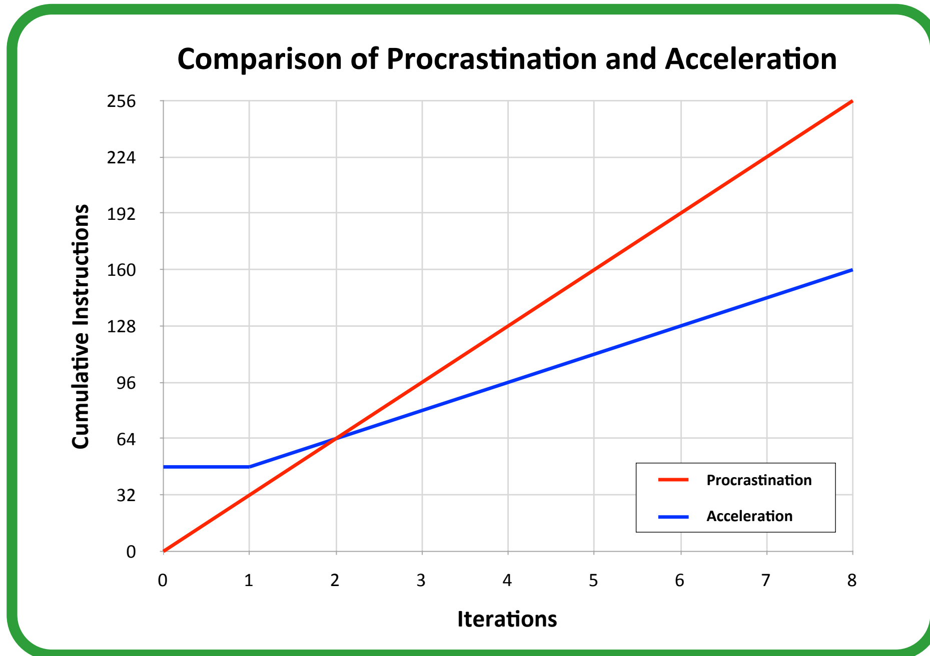

Figure 2: The red line is a linear function that represents the cumulative number of instructions using the conventional method of iterative computation, procrastination. The blue line represents the cumulative number of instructions when applying the differential concept of acceleration. To summarize, acceleration takes a number of extra instructions in the beginning but is much more effective in the long run.

```c
// Copyright © 2012-2013 George Morgan : All Rights Reserved
#include "arm.h"
#include "atmel.h"
#include "board.h"
#define BOARD_OSCOUNT           (AT91C_CKGR_OSCOUNT & (0x40 << 8))
#define BOARD_USBDIV            AT91C_CKGR_USBDIV_1
#define BOARD_CKGR_PLL          AT91C_CKGR_OUT_0
#define BOARD_PLLCOUNT          (16 << 8)
#define BOARD_MUL               (AT91C_CKGR_MUL & (72 << 16))
#define BOARD_DIV               (AT91C_CKGR_DIV & 14)
#define BOARD_PRESCALER         AT91C_PMC_PRES_CLK_2
void interrupt_handler(void) {
       while (1);
}
void init(void) {
       AT91C_BASE_MC -> MC_FMR = AT91C_MC_FWS_1FWS;
       AT91C_BASE_PMC -> PMC_MOR = AT91C_CKGR_OSCOUNT | AT91C_CKGR_MOSCEN; //Enable the main oscillator.
       while (!(AT91C_BASE_PMC -> PMC_SR & AT91C_PMC_MOSCS)); //Wait for the oscillator to enable.
       AT91C_BASE_PMC -> PMC_PLLR = BOARD_USBDIV | BOARD_CKGR_PLL | BOARD_PLLCOUNT | BOARD_MUL | BOARD_DIV;
       while (!(AT91C_BASE_PMC -> PMC_SR & AT91C_PMC_LOCK)); //Wait for the PLL to initialize.
       while (!(AT91C_BASE_PMC -> PMC_SR & AT91C_PMC_MCKRDY)); //Wait for the master clock to become ready.
       AT91C_BASE_PMC -> PMC_MCKR = BOARD_PRESCALER; //Switch to the slow clock prescaler.
       while (!(AT91C_BASE_PMC -> PMC_SR & AT91C_PMC_MCKRDY)); //Wait for the master clock to ready up.
       AT91C_BASE_PMC -> PMC_MCKR |= AT91C_PMC_CSS_PLL_CLK; //Switch to the fast clock prescaler.
       while (!(AT91C_BASE_PMC -> PMC_SR & AT91C_PMC_MCKRDY)); //Wait for the master clock to become ready.
       AT91C_BASE_AIC -> AIC_IDCR = B11111111; //Disable all external interrupts. *
       AT91C_BASE_AIC -> AIC_SVR[0] = (volatile unsigned int) interrupt_handler;
       #warning Ensure integrity of loop.
       for (int i = 32; i != 0; i --) {
          AT91C_BASE_AIC->AIC_SVR[i] = (volatile unsigned int) interrupt_handler;
       }
       AT91C_BASE_AIC -> AIC_SPU = (volatile unsigned int) interrupt_handler;
       AT91C_BASE_PITC -> PITC_PIMR &= ~AT91C_PITC_PITIEN; //Disable PIT interrupts. *
       AT91C_BASE_RTTC -> RTTC_RTMR &= ~(AT91C_RTTC_ALMIEN | AT91C_RTTC_RTTINCIEN); //Disable RTT intterupts. *
       AT91C_BASE_WDTC -> WDTC_WDMR = AT91C_WDTC_WDDIS; //Disable the watchdog timer.
       AT91C_BASE_SYS -> RSTC_RMR |= 0xA5000001; //Enable user reset.
       AT91C_BASE_PMC -> PMC_PCER = AT91C_ID_PIOA; //Enable the IO controller.
       //AT91C_BASE_MC -> MC_RCR = AT91C_MC_RCB; //Remap memory to RAM. *
}
```

Figure 3: The initialization sequence of the SAM7S in compliance with the locality principal.

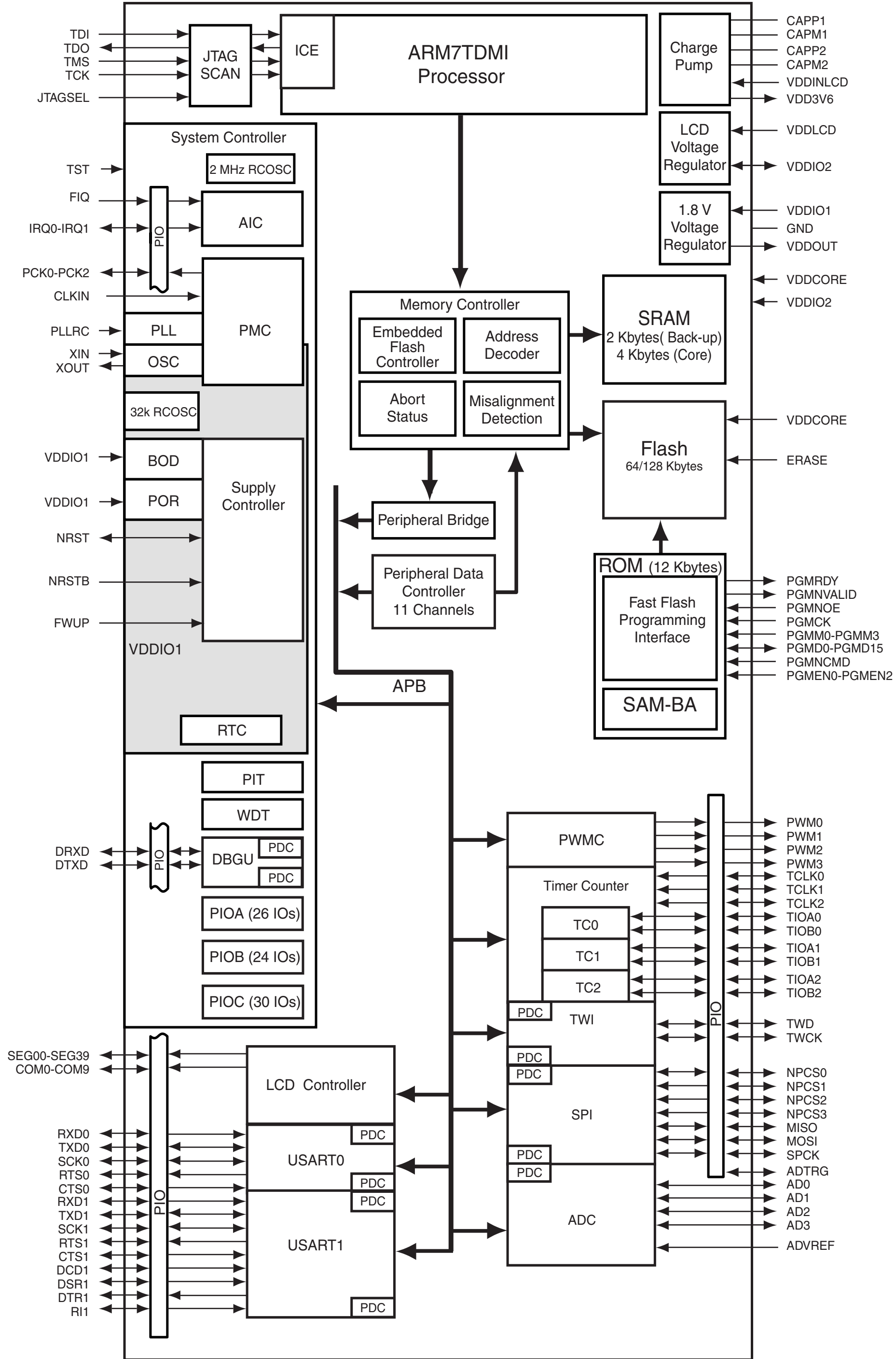

Figure 4: Block diagram of the SAM7S series of ARM processors.

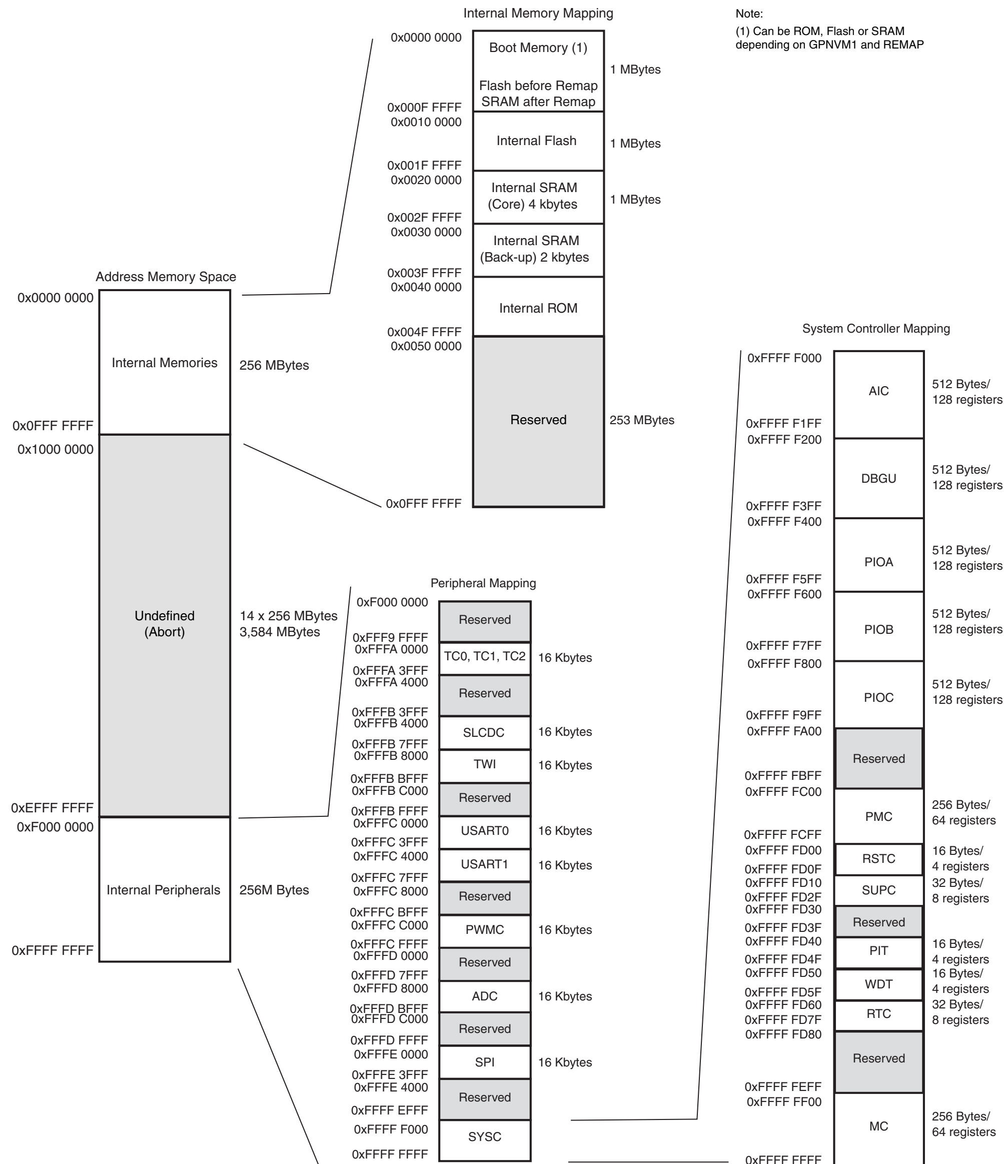

Figure 5: The extended peripheral map of the SAM7S.

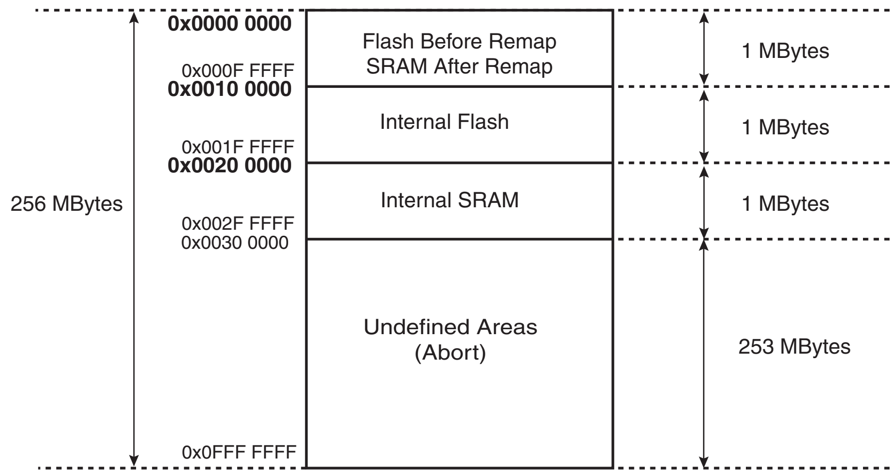

Figure 6: Internal memory of the SAM7S before remap.

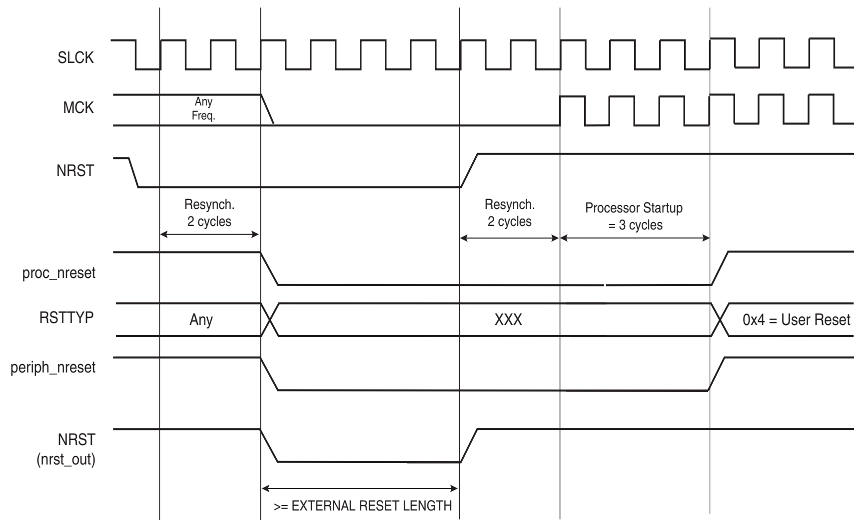

Figure 7: The number of cycles per iteration taken by the processor to initialize.

Figure 8: The relationship between all members of the UNIX family.