REVIEW AND TIMING ANALYSIS OF THE REAL TIME QP STATE MACHINE FRAMEWORK

By

CHRISTOPHER WEATHERS

Bachelor of Science in Mechanical Engineering

Oklahoma State University

Stillwater, Oklahoma

2009

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2014

REVIEW AND TIMING ANALYSIS OF THE REAL

TIME QP STATE MACHINE FRAMEWORK

Thesis Approved:

Dr. Gary Young

Thesis Adviser

Dr. R Delahoussaye

Dr. Prabhakar Pagilla

Name: Christopher Weathers

Date of Degree: July, 2014

Title of Study: REVIEW AND TIMING ANALYSIS OF THE REAL TIME QP STATE MACHINE FRAMEWORK

Major Field: MECHANICAL & AEROSPACE ENGINEERING

Abstract:

Embedded systems comprise the majority of all computer systems. Embedded systems require many considerations that general purpose computers do not. This is especially true of real time systems, which must reliably perform within exact parameters. Despite this, embedded systems have not been subject to as extensive an analysis in the literature from the viewpoint of someone wanting to develop an embedded product. Since failing to understand the characteristics required for developing embedded systems can cost money and even lives this is a problem. As such the specialized and esoteric nature of the information is readily available is insufficient for those looking to quickly and cheaply develop a product.

In this thesis the QP State machine framework is discussed. This is done for the goal of helping the reader understand both how the framework functions as well as why it is useful for developing real time embedded systems. Additionally, an attempt is made to make the concepts understandable from the perspective of a reader new to the intricacies of computing in embedded systems.

The specific implementation of the framework is done using only freely available software capable of running on an ordinary PC, as well as two Arduino UNO development boards. With the addition of a spreadsheet program, the timing characteristics of the framework are explored both on the theoretical and practical level with microsecond precision.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. REVIEW OF SYSTEMS AND PROGRAMMING STRATEGIES 9

System types under review 9
Coding Strategies 13
Sequential Control 13
Schedulers 17
UML Statechart Programs 23
What is a State? 24
Nested States 25
Extended States 25
Guard Conditions 26
Events 27
Actions and Transitions 28
Orthogonal Regions 32
Computational Limits of Statecharts 34

III. METHODOLOGY OF STATE CHART ANALYSIS 37

Timing Characteristics 38
Using the Arduino Uno as a Timer 38
Precision of Timing the QP Framework 40
State Machine Timing 42

IV. TIMING RESULTS AND DISCUSSION 57

Nesting Level 1 57
Nesting Level 2 59
Nesting Level 3 60

V. CONCLUSION AND RECOMMENDATIONS 65

Conclusions 65
Recommendations 69

REFERENCES 73

APPENDICES 75

Appendix A: Code 75
Appendix B: Nesting Level 3 Data 79
Appendix C: Lists of Origins and Destinations 83

LIST OF TABLES

Table Page

Table 1: Examples of system classifications 13
Table 2: Basic State examples 24
Table 3: Event, occurrence, and instance examples 28
Table 4: Initial Timer Test Results, n=9999 39
Table 5: Effect of Disabling Interrupts, 35us pulse n=9999 40
Table 6: Timing Assumptions and Consequences 42
Table 7: Origins, Destinations, and Transitions per Nesting Level 52
Table 8: Timing Results for Nesting Level 1 57
Table 9: Timing Results for Nesting Level 2 59
Table 10: Mean transition time for Nesting Level 3 61
Table 11: Mean Transition cycles for Nesting Level 3 62
Table 12: Reduced Dataset for Nesting Levels 1-3 64

LIST OF FIGURES

Figure Page

Figure 1: Categories of Systems under Review 10
Figure 2: Example Sequential Control Pseudo-code (left) and Flowchart (right) (Samek, 2008) 14
Figure 3: (A) Sequential Control (B) Scheduled (Samek, 2008) 17
Figure 4: Pre-emptive Priority Scheduling 22
Figure 5: Round-Robin with Priority Scheduling 22
Figure 6: Nested states (A) generic (B) toaster oven (Samek, 2008) 25
Figure 7: Fragile Keyboard (Samek, 2008) 26
Figure 8: Toaster oven state machine with entry and exit actions (Samek, 2008). 30
Figure 9: "UML state diagram of the keyboard state machine with internal transitions. (Samek, 2008)" 31
Figure 10: Orthogonal Regions of keyboard (Samek, 2008) 33
Figure 11: Naming Digit value vs. Location 44
Figure 12: Nesting Level Digits 44
Figure 13: Full Naming Convention Example 45
Figure 12: Prototype State Diagrams 49
Figure 13: QM State-Machine Example 53
Figure 14: Pulse Timing code in QF::onIdle() 54

LIST OF FORMULAS

Formula Page

Formula 1: Unique Origins 50
Formula 2: Basic Destinations 51
Formula 3: Non-Parallel Initial Transition Combinations 51
Formula 4: Number of Parallel Substates 51
Formula 5: Parallel Initial Transition Combinations 52
Formula 6: Total Unique Transition Destinations 52
Formula 7: Total Unique Transitions 52

CHAPTER I

INTRODUCTION

Today's world is one of rapid, exponential, change. This is especially true in regards to that uniquely human pastime of making tools, and applying those tools to every aspect of our lives. As a result, we are surrounded every day by the products of technology. Those products are increasingly complex. Yet those products are expected by those who buy and use them to be increasingly convenient. Paradoxically, we continue to expect new devices to be both improvements on previous generations and continue improving at the same rapid pace, month-by-month, year-by-year. Since the human capacity for thought is comparatively static, the approach to the dealing with the increasing complexity of development must be in how we think and the tools we use. Further, in order to better approach the problem, we must first understand the problem better.

Each day in 1985, the average person encountered around 3 micro-controllers. In 1990 that number had grown to 10. In 1995 the estimate was 50, from things like microwaves (1), cars (up to 10 as of 1995), and jets who might have a 1000 (Auslander, Ridgely, & Ringgenberg, 2002). As of 2009, automobile manufacturers were reporting an average of between 35 and 45 microcontrollers per car with luxury cars averaging 40 to 50, and BMW's 7 Series using up to 70 (Murray, 2009). If cars can be used as a benchmark for the number of microcontrollers encountered, a person may have encountered around 200 or more each day in 2009. Further complicating the issue is that as time goes on, newer chips are developed and deployed, and older

ones become obsolete. For instance the common brand Microchip, currently supplies 734 different micro-controller units. (Microchip Technology Inc., 2012) While some of these MCUs may be able to use identical code, many will not. Thus Microchip Technology Inc. can produce hundreds of distinct chips. Given that Wikipedia lists 37 different "common" brands (Wikimedia Foundation, Inc., 2012), the variety of different hardware platforms potentially numbers in the thousands today. That variety is likely to continue to grow as time goes on. Thus, developers must often choose a balance between the depth and breadth to which their development teams understand the hardware available. Choosing too much depth and too little breadth can result in suffering the limitations of the chosen hardware and forgoing the benefits of that not chosen. Too much breadth and too little depth can prevent fully exploiting used hardware as well as potential bugs due to poor understanding of the hardware's limitations. Both choices can be disastrous for a developer, whether a small one-person start-up, or a multi-national titan of industry. Strategies that allow for increasing both depth and breadth of understanding are thus of immense value.

Speaking of value, as computerized products become more ubiquitous, an increasing number of those products are high volume and or low cost. The low profit margins of low cost products, in tandem with the multiplicative effect of high volume products means that the cost per unit dedicated to the computational tasks is of critical importance. That cost can easily make the difference between an economical product and failed product. For instance Apple announced that, as of March 2011 they had sold approximately 108 million iPhones worldwide (Costello, 2012). That means that even a penny difference in the cost of the microcontrollers or microprocessors in the iPhone would have made a difference of over one million dollars. Thus it can be seen, although given a somewhat extreme example, that being able to efficiently utilize the resources of a given chip allowing for the use of less and cheaper hardware can make a huge difference in the viability of an endeavor. This means that a method of designing and developing software which avoids the use of excessive target hardware resources such as memory, RAM,

cycles, etc. is immensely more critical in embedded and low profile applications than it is in general purpose computing environments such as desktop and laptop computers. This is an important consideration as general purpose computers and microprocessors often have orders of magnitude more resources than embedded systems and microcontrollers, and as such often approach some programming tasks in an entirely different manner. Memory management is a good example as many programmers use heap memory, such as that provided by C language's functions malloc() and free(). This is a problem because the mechanism used is non-deterministic and often wastes memory through memory fragmentation (Walls, 2010). This is unacceptable to the resource-limited and often time-critical application of embedded systems and microcontrollers. Non-deterministic behavior is often worked around in general purpose applications by relying on the brute force of fast and power hungry microprocessors to handle the task in time. In the typically slower microcontrollers of embedded applications this strategy is less useful. In time-critical or safety applications, it is completely unacceptable to simply hope it will be done in time. Memory fragmentation is also likewise often ignored as the typical general computer has large amounts of RAM memory, and can often use hard-drive space as well. Even a slow continual leak of memory is often handled by restarting the computer. This often takes care of itself automatically because general-purpose computers are often restarted on a daily basis. An embedded system on the other hand may have only a few hundred bytes or kilobytes of available memory and may be expected to run undisturbed for years at a time, so ANY memory fragmentation might lead to a catastrophic failure costing money or lives. Speaking of brute force application of resources, the approach to designing and programming computerized systems must also take into account another important component of cost, time.

The time that it takes to develop a product is critical to the economics of the product. For one, there are the direct costs of time, wages of the personnel working on the project. Between the engineers, programmers, managers, and so on, personnel costs can accrue rapidly while

development is ongoing. Further, direct costs like wages are not the only costs associated with prolonged development time. A developer that can shorten their development times has a much greater range of strategic options regarding product and market forces. For instance, Widget Limited finishes primary development in 3 months, while their competitor Gadget-Ware takes 6 months. Widget Limited can choose to deploy first, gaining first mover status, and the associated advantages and disadvantages. Or Widget Limited might take the 3 extra months to test the waters, refine the product, and penetrate the market so as to have a superior product and customer image when both release. Other options are of course available and the subject of an extensive body of research, but Widget Limited has an undeniable strategic advantage over Gadget-Ware, due to having a greatly increased number of options. A further, indirect cost of development time, especially in software applications, is in product refinements. A development strategy that allows for rapid, yet predictable, change allows a developer greater responsiveness to flaws or shortcomings in products, reducing the impact of unexpected events during development. Further, if a developer is unable to respond in a fast and effective manner to a bug or flaw in their already released product, they may be passed over next time a customer searches for a supplier and can develop a nasty reputation. This ability to respond rapidly becomes more critical as the products become increasingly complex.

As the progression of home video players demonstrate, we expect our devices to take on an increasing set of tasks. VHS players were once considered perfectly functional with the ability to play, pause, stop, rewind, and occasionally to record. Then came the DVD player, with more features like a menu, cursor, scenes, bonus features, subtitles, multiple languages, director commentary, etc. Now we have a new generation of Blue-ray players, of which some even keep bookmarks or connect to the internet, accessing remotely stored features. Even further, these Blue-ray players are already facing competition in digital TV boxes which can stream shows and movies from services like Netflix or Hulu through an internet connection. Some of these devices

are blurring the line between televisions, stereos, pcs, and even phones. As these devices take on more tasks and features, they become increasingly complex. This increasing complexity can produce nightmares for developers if they do not have sufficient management strategies. This complexity means that even coming up with an exhaustive feature and specification list for a product can become a monumental task. Many development methods require such a list before development can even begin. Without the ability to efficiently partition individual aspects of a products behavior, clearly describing that behavior can be difficult. Without a clear idea of how a product behaves, implementing that behavior is made more difficult and a developer will have even larger hurdles for any new introduction or changes. This complexity and how a developer deals with it largely define how the development process progresses.

A common means of dealing with the complexity of a system is to distribute the system. This distribution is done by abstracting away groups of behavior into subsystems. This distribution takes place both in hardware and the development team itself. A car, for instance, may have different hardware controlling the engine, the brakes, the environmental controls, media devices, the lights, sensors, etc. Often these subsystems may have different people working on them, in parallel. Even on the software level, there are things like the user interface, hardware drivers, operating systems, applications, memory management, and various other distinct groups of behavior which need to be addressed. This distribution across both teams and hardware means that successful strategies must concisely and precisely define how each group of behavior interacts. These descriptions are important so that each subsystem can be the focus of its respective hardware and development sub-team. By describing each group of behavior both concisely and precisely, complexity can often be abstracted away. This allows the sub-team to effectively ignore the complexity of the overall system and focus on only one part of the system at a time. If the description is imprecise, the manner in which the subset interacts with the whole can be poorly understood. Poor understanding of these interactions can result in the system

failing to work together properly. Conversely, overly verbose descriptions of how each piece behaves, expose sub-teams to too much of the overall system complexity. This distracts from the concentration that can be applied to the individual subsystem, compromising performance. How strategies perform, describe, and implement this distribution of complexity can thus make the difference between a mind-numbing mishmash of concepts that break down development and hide bugs, to the ordered and efficient creation of a complex, yet deceptively simple and effective, product.

A consequence of partitioning product development across separate, sometimes parallel running, sub-teams is specialization. The brakes of a car, the fuel mixture of the engine, the suspension system, the transmission, the environmental controls, and so on are systems whose development and understanding often require highly specific, yet varied knowledge. Assigning someone with this detailed knowledge to each subsystem makes sense. However, requiring each team-member to have a detailed knowledge of computer theory, or a teammate who does, can be extremely prohibitive. Most engineers will not likely have both the detailed knowledge of an aspect of engineering as well as be comfortable delving into the intricacies of computer theory. As the writer has personally experienced while acting as a teaching assistant, the mindset of a skilled mechanical engineer is no guarantee of skill in programming. Computer science is a separate degree program for a reason, the considerations are almost endless. As a consequence, a development team may be composed mostly, or entirely, of members who have limited knowledge of computer programming, yet will be expected to program effective code. Simply requiring each parallel team to be watched over or include a computer scientist is cost prohibitive. Yet having insufficient understanding can cripple the subsystem. Thus how well a development team and its accompanying software programming strategy deals with this lack of familiarity can be critical.

As a product is under development, it needs to be tested. With the high complexity of products, it is often impractical to create a complete prototype for testing. Each piece of the system often has to have its individual performance tested, as well as how it interacts with other pieces of the system. One increasingly common and cost effective way of doing this is to simulate parts of the system. This allows different parts of the system to be tested individually and against other parts of the system without actually building them. As the capabilities of simulation software increases, this becomes an increasingly important part of the development process (Bartos, 2007). However in order to most effectively simulate a system, the software controlling the system needs to behave the same in both the simulation and the real system. One way of doing this is to use the same code during both testing and release. This is easier said than done though as many code strategies are very sensitive to hardware changes and have to be effectively rewritten to change hardware. Further, many coding strategies, such as super-loops, have to be extensively rewritten in order to implement new features. This extensive rewriting can radically change the behavior of the code, necessitating further testing. These portability and extension issues mean that a coding strategy which produces consistent, predictable behavior is important. By maintaining this behavior, while being both highly portable and insensitive to additions and changes, a strategy can greatly increase the effectiveness of simulation and other testing. Such effective simulation and testing can greatly reduce costs, speed development, and catch potential faults early.

As ever more products become increasingly computerized, reliability and safety become increasing concerns. The concern for reliability, and as a result safety, often trumps performance speed and efficiency. Take for instance the choice between an embedded controller that usually responds in a few microseconds, but might freeze or fail under rare circumstances, and one which will always respond in a millisecond or two. The former might be preferable in some instances like a graphics card, but not in car brakes or other safety critical systems. Even non-safety critical systems might prefer the latter, such as a sensor placed in a buoy out at sea or other hard to reach

systems. This need for reliability means that the strategy which produces the fastest code is not necessarily preferable to one which guarantees that the software will perform at a certain level. Of course, no coding strategy can prevent hardware failures. This need for reliability in both hardware and software is one of the reasons why otherwise obsolete microchips continue to be used. Older chips, having been field tested by previous products, tend to have more experimental data on failure modes and rates. This continued use of slower, older, and often power hungry chips means that guarantees are not a full substitute for the performance speed of code, but instead a constraint. Thus an effective coding strategy for the increasingly computerized world must take into account both efficiency and guaranteed performance simultaneously.

As we have seen, the changing face of technology means that the choice of coding strategy is crucial to the process of product development. A strategy has many demands that it must be measured against. For one, the sheer number of programming tasks necessary means that it must be applicable to a wide variety of applications. For another, the variety of hardware choices means that the strategy must produce highly portable code that is insensitive to hardware differences. The produced code must be compact in order to take advantage of the limited resources available. The strategy must be able to implement and refine code quickly. The strategy must aid the abstraction of the complex tasks it is presented with. The strategy must facilitate distribution of responsibility across development teams and hardware. The strategy must be easily understandable to a wide variety of disciplines. The strategy must also facilitate easy testing of both the complete system and subsystems, even before the product is built. And of course, any coding strategy that is to be used in systems which will be used in remote or safety critical systems must be above all else, reliable and guaranteed in its performance.

CHAPTER II

REVIEW OF SYSTEMS AND PROGRAMMING STRATEGIES

System types under review

In order to choose a strategy for development, current strategies must be reviewed and compared in their advantages and disadvantages. As we review these strategies, we must choose and remember our focus. For this paper, the focus will be on three types of systems which are becoming increasingly commonplace. The first of these types is embedded systems. The second type is reactive, or event-driven, systems. The third, and perhaps most restrictive, is that of real-time systems. These three types of systems are neither mutually inclusive nor exclusive, as shown in Figure 1 below, yet between them include a wide variety of systems used in many aspects of life. Each of these types of systems has their own considerations which must be addressed in their design and will be discussed below.

The first of the system types under review are embedded systems. While no strict definition of what constitutes an embedded system exists, the following can be said:

"A general definition of ***embedded systems*** *is: embedded systems are computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function. The word embedded reflects the fact that these systems are usually an integral part of a larger system, known as the embedding system. Multiple embedded systems can coexist in an embedding system" (Li & Yao, 2003).*

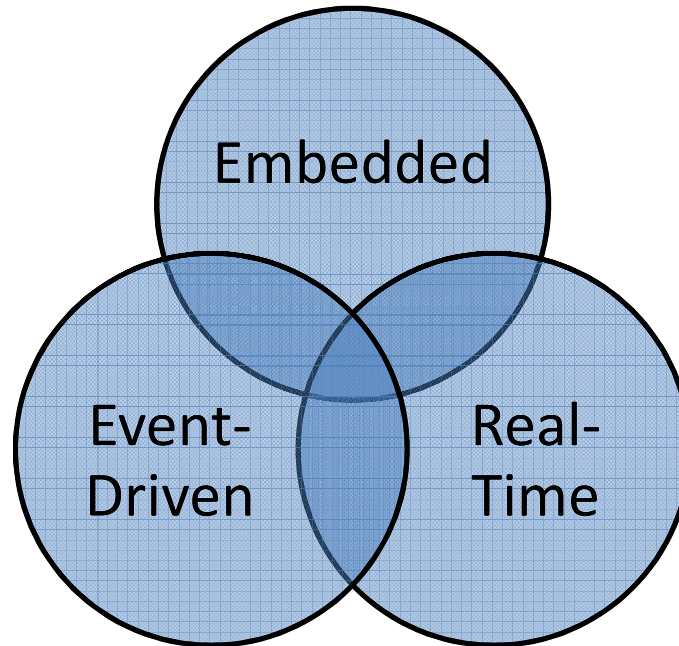

Figure 1: Categories of Systems under Review

Further characteristics typical of embedded systems are the parallel development of hardware and software as well as cross-platform development (Li & Yao, 2003). Parallel development refers to the practice of development teams for both the hardware of the system and the software to work together so as to take into account each other's advantages and limitations. By contrast, many hardware platforms such as PCs are developed with little concern as to the specific software applications which will run on them. Additionally, many of the programs which users are familiar with are not developed for a specific make and model of a computer, but rather a general purpose operating system like Windows. Cross-platform development, in turn, refers to a development process in which software crosses platforms from host to target. A platform, in this context, is a set of hardware, software, and development tools. The host platform is the one on which the program is developed. The target platform is the environment in which the program will actually run when deployed. This development approach is vital to embedded systems because they often run on resource limited hardware, and may not even have any user interface. The cross-platform approach thus allows for a more resource rich platform to be used by the

developer for the actual task of programming, like a PC with its keyboard, mouse, graphics, etc. As might be expected, these characteristics of embedded systems must be taken into account during the evaluation of software development strategies. As these embedded systems become increasingly common place, their importance as a topic of focus increases. At the same time, general purpose computers, as opposed to embedded systems, have been the subject of extensive research and development over the years. This disparity in research and development means that embedded systems are fertile ground for review.

The second type of system with which the author is concerned is that of reactive, or event-driven, systems. Event-driven systems are relatively common as many modern operating systems and Graphic User Interfaces (GUI) such as Windows are at least loosely event-driven. Samek gave a general description of the importance and characteristics of event-driven systems rather elegantly:

*"Almost all computer systems in general, and embedded systems in particular, are event driven, which means that they continuously wait for the occurrence of some external or internal event such as a time tick, an arrival of a data packet, a button press, or a mouse click. After recognizing the event, such systems react by performing the appropriate computation that may include manipulating the hardware or generating "soft" events that trigger other internal software components. (That's why event-driven systems are alternatively called **reactive** systems.) Once the event handling is complete, the software goes back to waiting for the next event" (Samek, 2008).*

Samek then goes on to compare this approach with that of sequential control, where the *"program waits for events in various places in its execution path by either actively polling for events or passively blocking on a semaphore or other such operating system mechanism"* (Samek, 2008). The problem, as related by Samek, is that "*while a sequential program is waiting for one kind of event, it is not doing any other work and is not responsive to other events*"

(Samek, 2008). It is this contrast in the approach to reacting to events which makes reactive systems event-driven. It is also what makes the ability of the programming strategy to deal with multiple potential events in an unpredictable order so important. Without the ability to respond in a logical, predictable, and efficient manner to events, the resulting system may have unacceptable performance behavior. Thus dealing with this behavior is an important characteristic of a truly wide-ranging software development strategy.

The third type of system under review is that of a real-time operating system, or RTOS. While a RTOS is closely related to an event-driven system, it is distinct. Further increasing the importance of a RTOS is that they are often used in embedded systems. In order to understand why this is the case we must first understand what a RTOS is and isn't. As summarized by National Instruments, *"real-time operating systems are designed to* ***run a single program with very precise timing****. Specifically, real-time operating systems can allow you to:*

- *Perform tasks within a guaranteed worst-case timeframe*
- *Carefully prioritize different sections of your program*
- *Run loops with nearly the same timing each iteration (typically within microseconds)*
- *Detect if a loop missed its timing goal" (National Instruments, 2012)*

In contrast to a real-time operating system there are systems running without an operating system, and those running general purpose operating systems. National Instruments stated this eloquently in that general purpose "*Operating systems like Windows are designed to maintain user responsiveness with many programs and services running (ensuring "fairness"), while real-time operating systems are designed to run critical applications reliably and with precise timing (paying attention to the programmer's priorities)" (National Instruments, 2012).*

These three types of computing systems are often intertwined and related, but are nonetheless independent types. As shown in Table 1 below, a system can be many combinations thereof. Of course due to differences in implementation, two otherwise similar systems might be categorized differently. As might be seen by the examples, real-time operating systems are rarely used outside out of embedded systems. This is because their tendency to focus on running small groups of tasks with specific timing means that they are almost always dedicated systems with tight hardware interaction and are thus are also embedded systems.

Description of System	Embedded	Event-driven	Real-time
Old PC	No	No	No
Alarm Clock	Yes	No	No
Modern Windows PC	No	Yes	No
Development PC running non-event-driven RTOS	No	No	Yes
Development PC running event-driven RTOS	No	Yes	Yes
Touchscreen	Yes	Yes	No
MRI machine	Yes	No	Yes
Modern Car Brake Controller	Yes	Yes	Yes

Table 1: Examples of system classifications

Coding Strategies

There are many strategies as to how to organize the various operations performed by a program. Each of these strategies has various advantages and disadvantages. Depending on the nature and requirements of the system, the optimal strategy for use can vary. Below is a discussion of some strategies for organizing operations. As these strategies are more or less extensions and specialized cases of each other, the simplest case will be discussed first.

Sequential Control

The most basic of strategies for organizing the operations of a program, is sometimes referred to as **sequential control**. This approach for handling the execution of program operations is more a lack of strategy than strategy. This means that sequential control is mutually exclusive with

event-driven or real-time operating systems. This strategy consists of building the program order directly into the outermost main loop as shown in Figure 2 below.

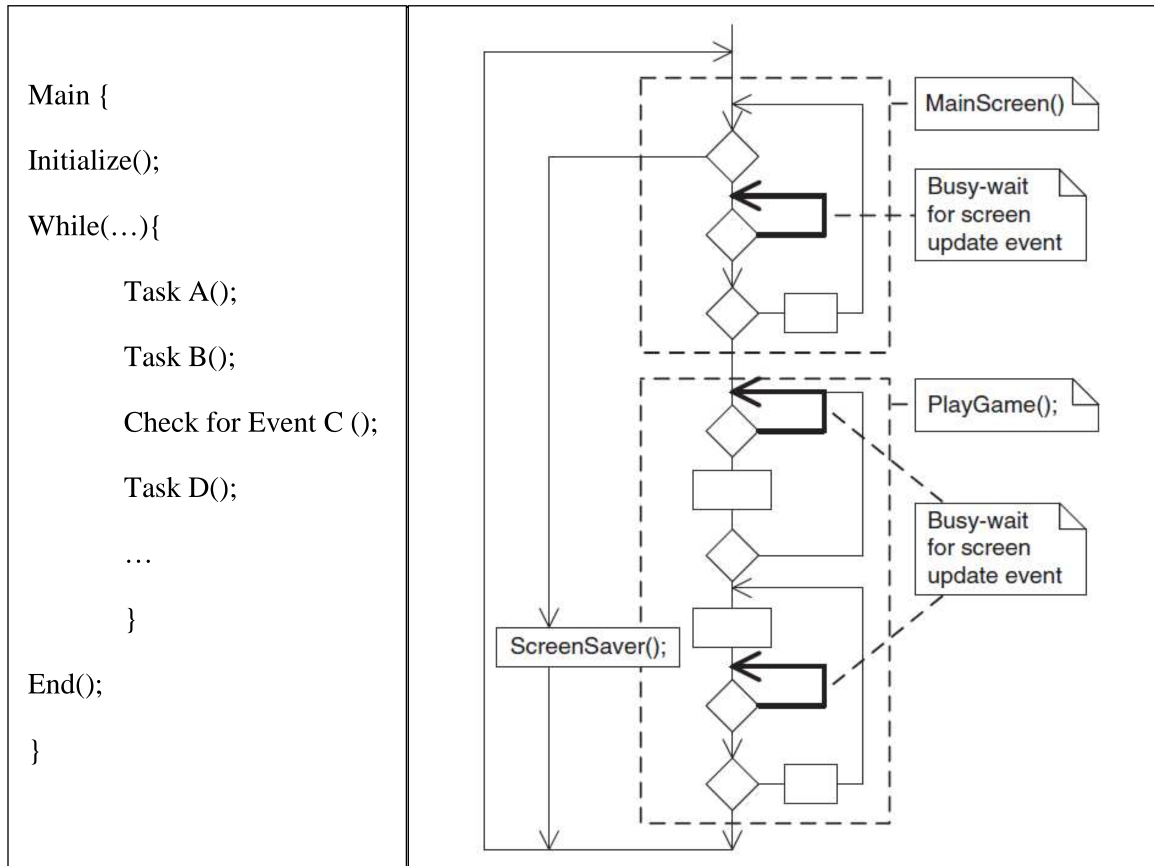

Figure 2: Example Sequential Control Pseudo-code (left) and Flowchart (right) (Samek, 2008)

While in some simple programs the sequential control approach can be simple and effective, it has many disadvantages. Event responsiveness can suffer greatly due to the fixed order of code execution. Further, the pre-scripted order in which all tasks occur can result in bloated and difficult to follow code. Lastly, getting task timing precise can be difficult. These difficulties grow rapidly as more code and tasks are added.

As a program takes on more tasks and events which it must deal with, event responsiveness can suffer. In sequential programming this can be particularly extreme. By fixing the order of program execution, the program can only attempt or wait for one task or event at any given time. If a program is not ready for a task when code execution reaches it, the program must either skip

the task entirely or wait until the task can proceed. If the program skips the task, then the program is not able to attempt to perform the task again until the code loops all the way around again or otherwise explicitly checks readiness. This can mean that missing the conditions by a single clock cycle has the same effect on responsiveness as missing by an entire iteration of the program. Alternatively, the program may choose to simply wait until it is ready for the task or the event occurs. This approach however means that the program can respond ONLY to that task or event, and any others must wait until the designated event or condition occurs. Waiting for a relatively unimportant task while being unresponsive to critical events has obvious downfalls, and is likely unacceptable in safety or performance-critical applications. Limiting the time for which a program will wait for an event to occur can provide some relief from this problem. Additionally, a program may check if a task is ready or an event has occurred in multiple, strategically checked places. However, limited wait times and repeated conditional evaluations can rapidly bloat and obscure the flow of code, as well as wasting valuable system resources.

As the number of tasks and events that a sequential control has to handle increase, the complexity of the code to handle execution order grows rapidly. In order to pre-script the order in which a program executes, conditional statements are often necessary for circumstances in which multiple avenues of code execution must be chosen from. This is done because a programmer must try to take into account all of the valid code paths which can occur during operation. As a result, each possible path, or similar group of paths, tends to get at least a small amount of code to handle it. These conditional statements of code execution tend to result in an abundance of nested conditional statements, variables to store context, repeated code, or small functions. It is easy to forget to deal with all possible results of nested conditionals, and properly assign and update all context variables for each case. Repeated code both uses extra resources, as well as providing ground for a single bug to reappear throughout the code, easily resurfacing after it is thought to be fixed. A common solution to this repeated code is to wrap them up into a new function. This,

however, can result in a large number of small functions with obscure purposes. Further, between the extra function calls and variable passing, this can come with extreme consequences in terms of performance, memory, and stack usage. All of these issues obscure the code and provide fertile ground for bugs to occur. Careful organization can keep the extra code to a minimum and somewhat readable. However, as more and more decisions must be made and paths become more complex, code size begins to grow rapidly. Further, and likely more critical, it becomes more and more difficult for the programmer to follow the resulting maze of obscure and bloated code.

Even without obscure and bloated code, sequential control tends to make precise timing difficult. One of the main reasons for this obscuration, bloating, and difficult timing is that task timing is heavily interdependent. Timing interdependence is due to the pre-scripted order in which tasks occur. Each task's timing is directly dependent on each preceding task. For example, if, in a program running (ABCABC…) task A has its code changed, all three tasks have altered their timing. This means that code changes and additions have a cascading effect on timing. This cascading effect can obviously make code maintenance difficult. This is especially difficult if code can take more than one path, as each might have unique timing. Of course one way around this is to periodically have a task wait for some timer to trigger. This allows you to slow down and fix the frequency by calibrating to the timer. This is only a stopgap method however as while one task is waiting for its timer, the program is effectively frozen. This (A-wait-B-wait-C…) or (ABC-wait-ABC-wait…) approach can quickly use up available clock cycles and timers. This also requires lots of excess speed to insure that each task finishes before the wait time for the next is ready. Furthermore, timers are generally precious resources and using them in this manner tends to exhaust them quickly. As more tasks are added the complexity of this increases exponentially. In multi-rate systems, this can become nearly impossible. Once you start sharing timers between multiple tasks to make things simpler, you are either on your way to getting spaghetti code or are leaving sequential control behind.

Schedulers

In order to leave sequential control behind it is necessary to start delegating responsibility for the order of the execution to the code itself. This is done by having the outer loop of the program running an algorithm which chooses which code to execute. This is in contrast to sequential control where the outer loop runs the code in a predetermined fashion. Figure 3 provides an example side-by-side comparison of a sequential program and an event-driven scheduled program. The algorithm, and its accompanying code, is called the **scheduler**.

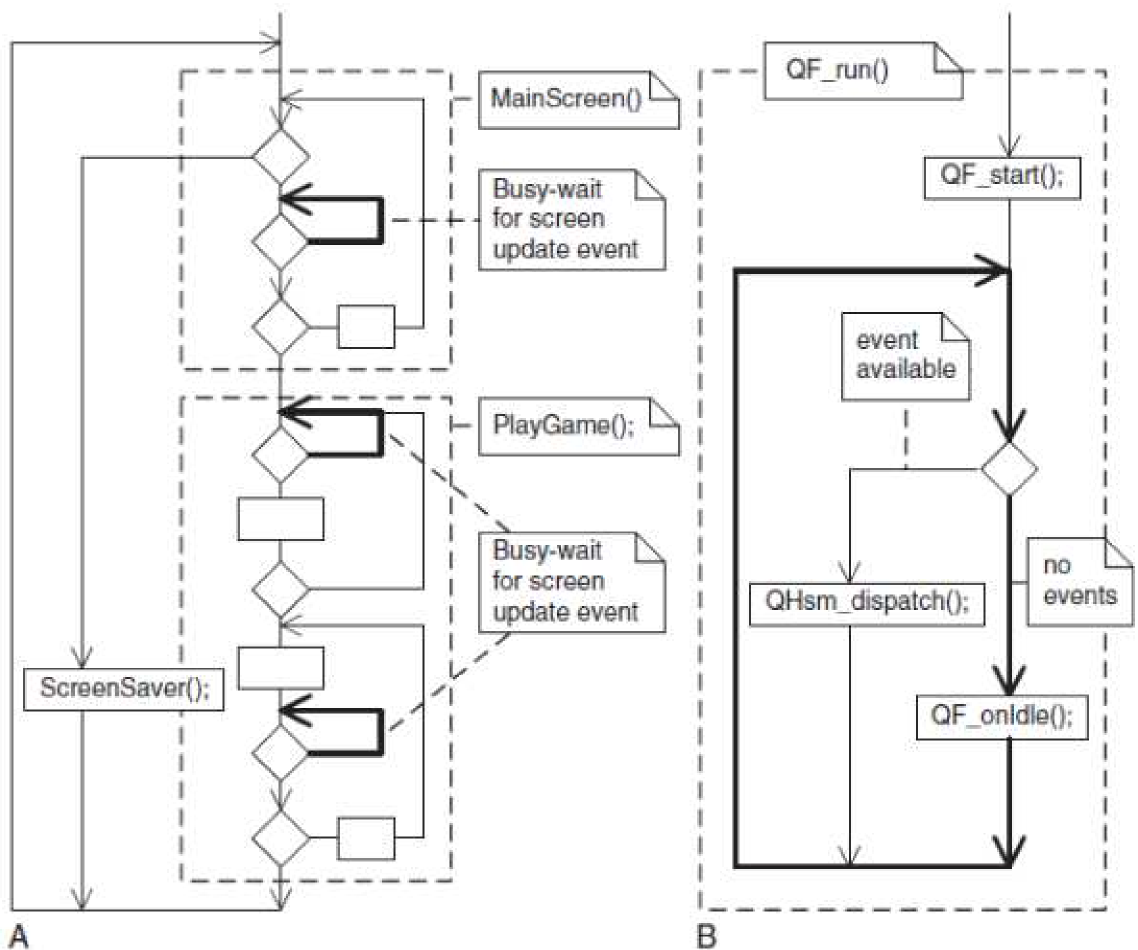

Figure 3: (A) Sequential Control (B) Scheduled (Samek, 2008)

The scheduler is the portion of the program that chooses which code sections to run and when. In turn "*The scheduler is at the heart of every kernel*" (Li & Yao, 2003). A kernel is the

fundamental core of an operating system. Different kernels vary in the services they provide, but any multitasking operating system kernel requires at least the use of a scheduler. As the heart of the kernel, the scheduler deserves further discussion. To provide the appropriate background information, two important hardware concepts must be discussed first.

The first of these concepts is the stack. The stack is where the return address is stored when a function is called. This is necessary so a chip knows where to continue execution after a function or call returns. Based on the hardware, programming language, and implementation, a stack may also store a variety of other information like local variables, register values, etc. Some hardware implementations have a special hardware-based stack with fixed size, only accessible from the top. An example of this is the PIC18f4520 from Microchip with room for 32 return addresses and no other data (Microchip Technology Incorporated, 2008). On the other hand, some implementations use the general data SRAM allowing for varying size as well as random access. An example of this is the ATmega328P, which places the stack in the 2KB general memory allowing great flexibility in its use (Atmel Corporation, 2012). The ATmega328P is one of the microprocessors used in the Arduino. This is important in a multitasking environment because the size and accessibility of the stack has a great deal of influence on the ease and manner of managing multiple running tasks. This has a great deal of effect on the feasibility of various scheduling algorithms, making them somewhat dependent on hardware.

The second of these hardware concepts that needs to be discussed is that of the interrupt. *"Basically, an interrupt causes a program to suspend its current operation and branch to a location elsewhere in memory. Then, after the program handles the event that caused the interrupt, the interrupt service routine (ISR) must restart the program from where it had previously been suspended* (Rosenthal, 1995)." This is akin to closing a book around a pencil to keep your place in order to respond to someone's unexpected voice. You were reading without constantly checking for a voice, but when your ear detected the noise, it interrupted your reading.

Then when the persons comment has been handled, you can return to where you left off thanks to your pencil saving the place. Different chips have different interrupts, but typically include three primary sources. These interrupts are triggered by events external to the CPU, certain software errors (like divide by zero), and certain special instructions. An interrupt allows a program to note and respond quickly to an event without needing to specifically check or wait for the event. Since normal code execution is suspended while in an interrupted state, interrupts must follow two basic rules: "*First, the ISR must save then restore all CPU, memory and I/O resources that it uses. … Second, it must get back out of the ISR as quickly as possible. The reason for this rule is that ISRs should generally block new interrupts until after the ISR has completed running. Therefore, an ISR should do as little as possible so that interrupts are off for as little time as possible* (Rosenthal, 1995)*."* The first rule is important because otherwise, the interrupt service routine might corrupt data being used by the code it interrupts. Many hardware devices are designed to streamline the first rule by automatically saving and restoring many of the necessary resources with special instructions. Interrupts provide a large part of the framework which allows CPU peripherals and other external devices to communicate with CPU itself when they are ready. As a result, CPU clock cycles are only used when the external world comes knocking. As such interrupts are the means by which the CPU, and in turn program, can efficiently get information from the external world, critical for event-driven systems.

Now that some of the background concepts have been briefly explored, we can discuss the scheduler itself in more detail. A brief overview of a scheduler is as follows. The scheduler is responsible for allocating CPU time to **schedulable entities**. When more than one schedulable entity exists, the scheduler is performing **multitasking**. When a scheduler changes from one entity to another, it is performing a **context switch**. The set of rules a scheduler uses to allocate time is called a **scheduling algorithm**. Once the scheduler has made its decision, it uses a

section of code called the **dispatcher** to implement the decision. This overview is necessarily sparse and needs to be expanded upon further.

One of the first concepts to be expanded upon is what exactly is scheduled. The entities that are handled by the scheduler go by various names and have varying properties depending on the kernel and implementation. The most basic of these entities is the task, also sometimes called threads. "*A task is an independent thread of execution that contains a sequence of independently schedulable instructions* (Li & Yao, 2003)." Essentially, this makes tasks individual sequentially controlled programs. Tasks also contain a small amount of information for helping the kernel and operating system to keep track of what it needs to operate on the task. The exact resources the kernel assigns to a task necessarily vary greatly depending on the individual kernel. Another common entity is that of the process, which is an essentially more feature-rich task supported by some kernels. A process has more resources assigned to it and in some kernels can contain multiple threads and tasks within itself to be scheduled. Maintaining multiple tasks is what allows a scheduler to perform multitasking.

Since a single CPU core can only run one set of instructions at a time, only one task can actually run on a core at a time. By switching between tasks a scheduler is said to be multitasking. The scheduler is allowing each task to run as if it were running alone on the CPU for a limited time before switching to another task. When this switching is done quickly, it can lead to the illusion that multiple tasks are running simultaneously. In some kernel-hardware combinations, a scheduler allows for allocating tasks across multiple CPU cores, allowing for multiple tasks to truly run at the same time, managed by the kernel's scheduler. Multitasking has pitfalls however, as it introduces a whole world of potential problems. Most of these problems revolve around the use of shared resources and concurrency issues. The act of switching control from one task to another is what causes many of these issues.

The actual act of switching from one task to another is called the context switch. This context switch needs to save the context of the task being stopped and restore the context of the task being restarted. Depending on the kernel and hardware implementation this context information includes: the status of various CPU registers, the stack used by the task, as well as various other bookkeeping information. When done properly, the task itself has no knowledge of the switch occurring, operating as if it had run alone and undisturbed. Because of the bookkeeping information required, exactly how a context switch is performed is often heavily processor specific. Additionally, the context switch entails overhead, and can degrade performance when an application is designed to include frequent context-switching. Thus, when a context switch happens and what task to run next are important considerations.

When a context switch can occur and how the next task is chosen is what constitutes the scheduling algorithm. One of the most important questions to ask of a scheduling algorithm is whether it is pre-emptive or cooperative. In a kernel running a cooperative scheduling algorithm, context switches only occur as a result of explicit calls to the kernel by the running task. This means that if the running task does not use a kernel call which can result in a context switch, no other tasks will run. In contrast, a pre-emptive kernel also generates context switches from interrupts, such as timers or peripherals (Samek, 2008). This means that a context switch can occur at any point during the execution of a task, unless explicitly disabled. A cooperative kernel control is passed much like the baton in a relay race, or a game of hot potato. On the other-hand, a pre-emptive kernel can forcibly take away control from a task by interrupting its execution, like a fumble or interception in sports, or the changing the channel in the middle of a TV show. Another question to ask of a scheduling algorithm is how it chooses which task to run when a context switch occurs.

Two basic means of choosing which task to run next are Round Robin and Priority (Li & Yao, 2003). In Round Robin scheduling, processes are run one after the other in order. True round-

robin algorithms are pre-emptive systems where tasks are given specific time-slices and interrupted after their allotted time is up, whether finished or not. Then once all other tasks have had their turn, control is returned to the interrupted task. In a priority based system, tasks are each assigned a priority. When a context switch occurs in a priority based system, the highest priority task ready to run is then chosen to run next. In a pre-emptive environment, a context switch occurs whenever a higher-priority task becomes ready, interrupting the current task. Many other methods exist as well as hybrid methods. Figures 4 and 5 below show two scheduling algorithms commonly used in real-time systems (Li & Yao, 2003). The full discussion of different possible algorithms and their implementations, advantages, and disadvantages is far beyond the scope of this paper, and quite possibly an entire textbook.

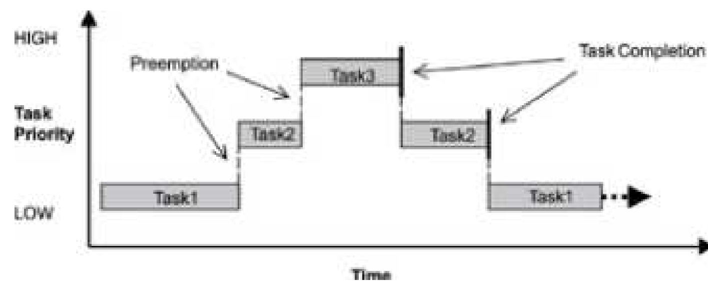

Figure 4: Pre-emptive Priority Scheduling

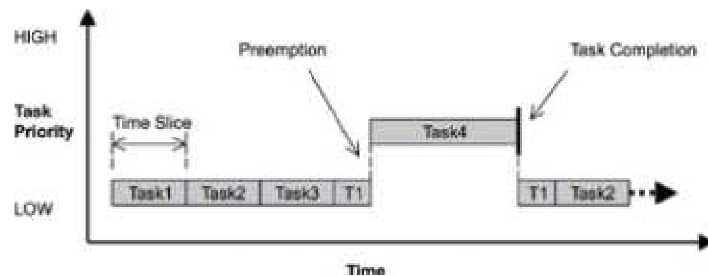

Figure 5: Round-Robin with Priority Scheduling

Of course once a scheduler has decided which task to run next, it must actually pass control to that task. This passing of control and the accompanying context switch is performed by the dispatcher. The dispatcher is what is called whenever control is in the kernel and the kernel is ready to pass control back to the user's application. Special attention must be paid to interrupt

service routines. This is because the dispatcher cannot be called during the execution of an interrupt service routine. In cooperative systems, the dispatcher is not called at all during an interrupt service routine. In pre-emptive systems, the dispatcher is called as the interrupt service routine exits. It is this calling of the dispatcher after an interrupt service routine which allows for pre-emption. The actual implementation of the dispatcher varies based on hardware, kernel, and scheduling algorithm.

Beyond the scheduling algorithm, how a kernel functions can have an impact on how tasks are programmed. The way the tasks themselves are structured is in itself an important part of how the system behaves. Any time you use any form of multitasking, the structure of the tasks and how they interact with each other and the kernel is important. How this interaction is done is much of what defines how a program interacts with the outside world.

UML Statechart Programs

There exist a large variety of fairly traditional multitasking kernels, such as Linux. Each can and often is the subject of volumes of work on how to utilize the kernels tools to code multitasking programs, and the pitfalls to be avoided. To explore any one in detail would be a large undertaking, and can be daunting for a programmer. This is largely because such traditional tools often rely heavily on the programmer to structure the program and code using the tools in such a way as to deal with concurrency and resource sharing issues. As such we are now focusing instead on a different strategy which underlines the approach which the author intends to explore in later chapters. This strategy is to organize the desired program into a set of state machines through the use of Unified Modeling Language, or UML, Statecharts. Of course in order to explain the method and benefits of organizing a program as a state machine, UML Statecharts must first be explained.

What is a State?

At heart, a statechart is a diagram of **states** defining a state machine, their identifying properties and how they interact. So what exactly is a state? A state is essentially a "*chunk of behavior*," wherein the object or program behaves in a certain way (Samek, 2008). The idea being that the state of a program is the only relevant information needed to determine how to respond to any given input. For a list of examples, see Table 2 below. A state captures the relevant aspects of the system's history very efficiently. For example, as far as a keyboard is concerned, the set and order in which keys have been pressed in the past don't matter so much as whether caps lock, ctrl, and shift are active. A state can abstract away all possible (but irrelevant) event sequences and capture only the relevant ones. This means that instead of recording the event history in a multitude of variables, flags, and convoluted logic, as is the traditional approach, you rely mainly on just one state variable that can assume only a limited number of a priori determined values, such as on or off, heating or cooling, etc. The value of the state variable crisply defines the current state of the system at any given time. The concept of state reduces the problem of identifying the execution context in the code to testing just the state variable instead of many variables, thus eliminating a lot of conditional logic. Moreover, switching between different states is vastly simplified as well, because you need to reassign just one state variable instead of changing multiple variables in a self-consistent manner (Samek, 2008).

Description of System	States
Microwave / Oven	On, off, heating, baking, door open, etc.
Keyboard	Capslock_on, CapsLock_off, ctrl_pressed, etc.
Car	Off, Drive, Reverse, Park, etc.
Motor	On, off, Stalled, Overheating, etc.
Time bomb	Set, Disarmed, Countdown, Detonated
Seatbelt	Latched, unlatched

Table 2: Basic State examples

Nested States

In many even moderately complex systems, there are often states which are very similar, or even overlap. For example, a toaster oven might have a baking state as well as a toasting state. In this case, both states also imply being in the heating state. Rather than define two completely separate states with duplicated behaviors, UML statecharts allow for the hierarchical abstraction of common behavior. This is done through the use of hierarchically nested states, making the statecharts *hierarchical state machines* or HSMs. Figure 6 below shows an example of how that looks, using the toaster as an example.

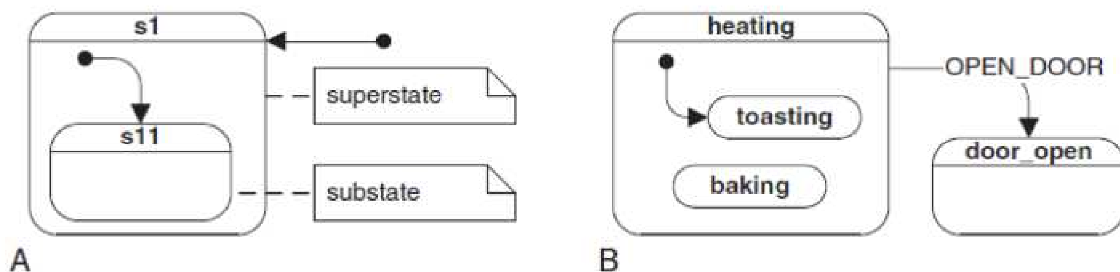

Figure 6: Nested states (A) generic (B) toaster oven (Samek, 2008)

The use of nested states allows for common behavior to be grouped in higher level states. Further nested states can then be "programmed by difference." This is similar to the inheritance of object oriented programming, where the "is-a-kind-of" relationship of inherited objects is replaced by the "is-in-a-state" relationship of nested states. This abstraction allows for the developer to zoom in and out as necessary to the needed level of detail. Further, with extended states, nesting helps to prevent the phenomenon of state explosion whereby the number of states necessary to describe a system increase geometrically as the complexity of the system grows (Samek, 2008).

Extended States

Since having a different state value for each and every situation would create a vast number of states for a program with even something as simple as a counter (a 8 bit counter could add 256 states, two would add 256*256=65536 states), program variables are often separated from states.

Rather, the complete condition of the system (called the **extended state**) is the combination of a qualitative, behavioral aspect (the state) and the quantitative, data storage aspects (the extended state variables) (Samek, 2008). In this interpretation, a change of variable does not always imply a change of the qualitative aspects of the system behavior and therefore does not lead to a change of state. State machines supplemented with variables are called extended state machines and UML state machines belong to this category. As an example, in Figure 7 below is an example statechart modeling a keyboard that breaks down after 1000 key presses. The advantage of this is that changing the durability is only a matter of changing a single variable, rather than changing the states themselves.

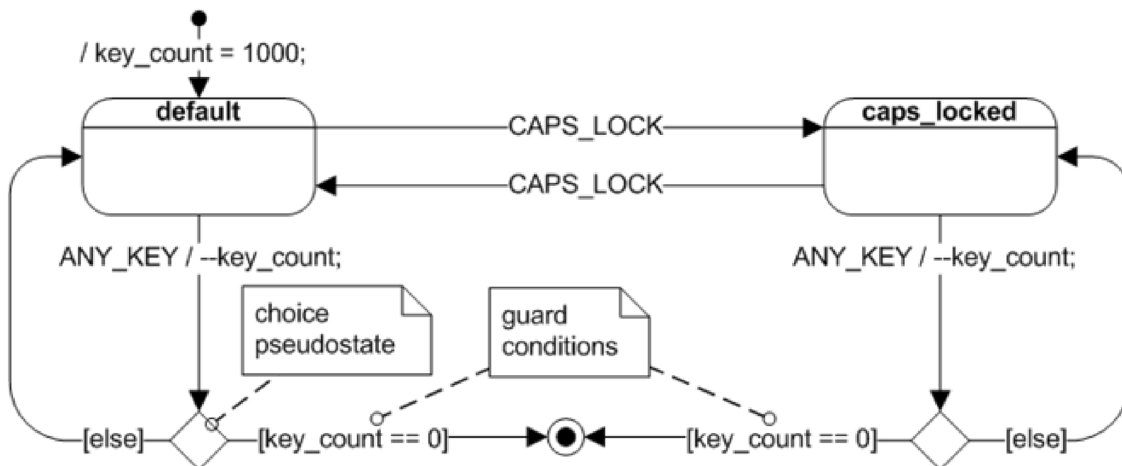

Figure 7: Fragile Keyboard (Samek, 2008)

Guard Conditions

The introduction of extended states is of little use without a means of them influencing the behavior of the state-machine. One of the more important mechanisms by which extended state variables can be used to control the behavior is through the use of guard conditions. To summarize, a guard condition is essentially a true/false statement which is evaluated to see which path a state transition will take or if an action will fire. As an example, the fragile keyboard from Figure 6 tests whether or not there are any key presses left each time a key is pressed in order to

determine whether to break down and enter the final state, or continue operation. Without guard conditions, extended state variables would not be able to control state-transitions, thus rendering them largely useless. Of course care must be taken that guard conditions are not over-used. Such abuse is "*the primary mechanism of architectural decay in designs based on state machines. Usually, in the day-to-day battle, it seems very tempting, especially to programmers new to state machine formalism, to add yet another extended state variable and yet another guard condition (another if or an else) rather than to factor out the related behavior into a new qualitative aspect of the system—the state* (Samek, 2008)."

Events

Now that we have talked about States and guard conditions, we might be asking ourselves: How do we pass information to and from the state machine, how do we let it know something happened that it needs to respond to? To do this, we use **events**. "*In the most general terms, an event is an occurrence in time and space that has significance to the system* (Samek, 2008)." It is also worth noting that this does not strictly mean that the event must be a physical event, it could also be the same or another state-machine announcing that it has done something, etc. In discussing events some terminology must be fleshed out.

The three main terms relevant to events are **event, occurrence,** and **instance**. The UML specification states that "*An event is the specification of some occurrence that may potentially trigger effects by an object* (Object Management Group, 2011)." In other words this means that **event** refers to a type, **occurrence** refers to an individual happening, and **instance** refers to a specific event-occurrence pair. For example, pressing a power button at noon on Friday is an *instance*. This instance is in turn composed of the *event* the instance is a type of, a power button press, and a specific *occurrence* of that event, the one Friday at noon. This may seem to be unnecessarily complicated, but each of the three is a pivotal source of information.

The requirement of the three terms can be seen in the processing lifecycle of event instances. First the occurrence of an event is an instantaneous thing which the system must process. This need for response causes the need to create an instance recording what type of event occurred and any other relevant parameters which might be needed to process, such as which button was pressed. Once this instance has been created it might be conveyed to one or more state-machines for processing. This means that the instance necessarily outlives the actual occurrence and may linger in the system for some time before it is processed and consumed. It is quite possible for another occurrence of the event to occur before all previous occurrences have been processed, thus the need for an instance for each occurrence, rather than each event. By separating information common to each occurrence into the specification of the event, redundant information can be reduced. Given the need for more specificity in technical processes rather than general conversation, this can be rather confusing and take time to get used to. Hopefully, Table 3 below and the actual implementation and use of events in chapters 3 and 4 will help to make it more understandable.

	Event	Occurrence	Instance
1	KeyPress	'a'	KeyPress: 'a' at 'b'
2	StateEntered	'on'	StateEntered: 'on' at time 'b'
3	Error	'divby0'	Error: 'divby0' at line 'c' at time 'b'
4	Message	'hello'	Message: 'hello' at time 'b' in memory location 'c'

Table 3: Event, occurrence, and instance examples

Actions and Transitions

Now that we have objects to act on (states and extended state variables), ways to inform them (events), and means of making decisions (Guard conditions), we need something to actually do if statecharts are to be a useful programming strategy. Once a state machine has received an event and the guard conditions have been evaluated, the state machine will then perform an action

and/or transition. Actions, in this context can refer to various things, "*such as changing a variable, performing I/O, invoking a function, generating another event instance, or changing to another state. Any parameter values associated with the current event are available to all actions directly caused by that event* (Samek, 2008)." When an action causes a change in state, the process is called a **state transition** or transition for short. UML statecharts have a variety of special types of actions and transitions which warrant further explanation.

Two very important classes of actions, especially for HSMs (Hierarchical state machines), are entry and exit actions. Entry and exit actions are performed whenever a state is entered or exited, respectively. Because these actions are associated with a state itself, rather than a transition, "*they often determine the conditions of operation or the identity of the state, very much as a class constructor determines the identity of the object being constructed* (Samek, 2008)." There are three major advantages of associating entry and exit actions with the states themselves. First is a reduction in redundancy. Hierarchical states mean that any given transition might pass through a variety of states on its way from source to target, each of which could have actions that need to be performed as they are entered or exited. Additionally, any given state, might have multiple transitions associated with it. Without entry and exit actions, the developer would have to ensure that each and every one of these transitions performed the correct actions as it maneuvered between the various states. This could easily result in duplicate and redundant code. A second advantage "*of entry and exit actions is that they provide means for guaranteed initialization and cleanup, very much like class constructors and destructors in OOP* [Object-Oriented Programming] (Samek, 2008)." This is important both for preventing errors as well as helping to define the state. As an example of how entry and exit actions can help to prevent errors and define the state, see Figure 8 of a toaster oven below. In this case, it can be seen how the superstate "heating" is defined by having the heater on, the "door_open" state by having the light on, and the two heating substates by their respective settings. In each case, the entry and exit

actions ensure that these identities are preserved. The third advantage is that as a result of the reduction in redundancy and preservation of identity, adding or altering states and transitions becomes much simpler and less error prone, aiding the maintainability of the software.

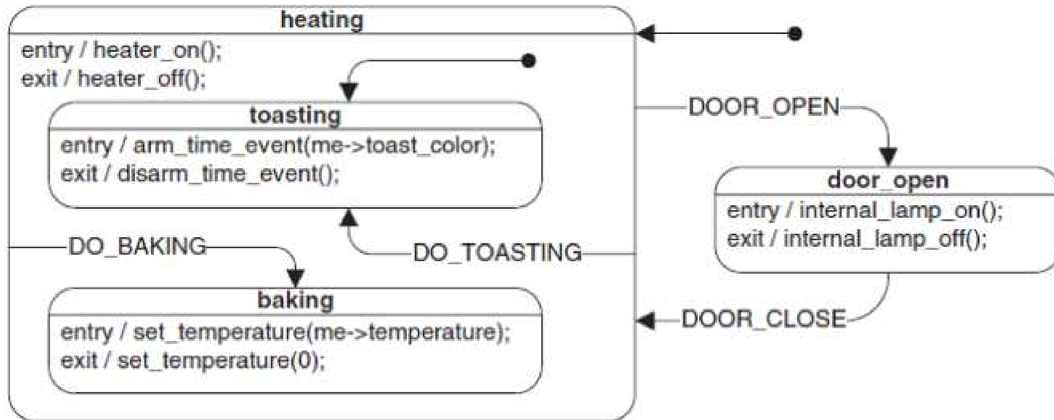

Figure 8: Toaster oven state machine with entry and exit actions (Samek, 2008).

A second class of actions worth mentioning is those which do not cause a change in state. These actions are also referred to as *internal transitions*. The important aspect of internal transitions to note is that "*no entry or exit actions are ever executed as a result of an internal transition, even if the internal transition is inherited from a higher level of the hierarchy than the currently active state. Internal transitions inherited from superstates at any level of nesting act as if they were defined directly in the currently active state*. (Samek, 2008)" For an example of how internal transitions look, see Figure 9 below, of a keyboard which responds to any key press, but only changes state with CAPS_LOCK.

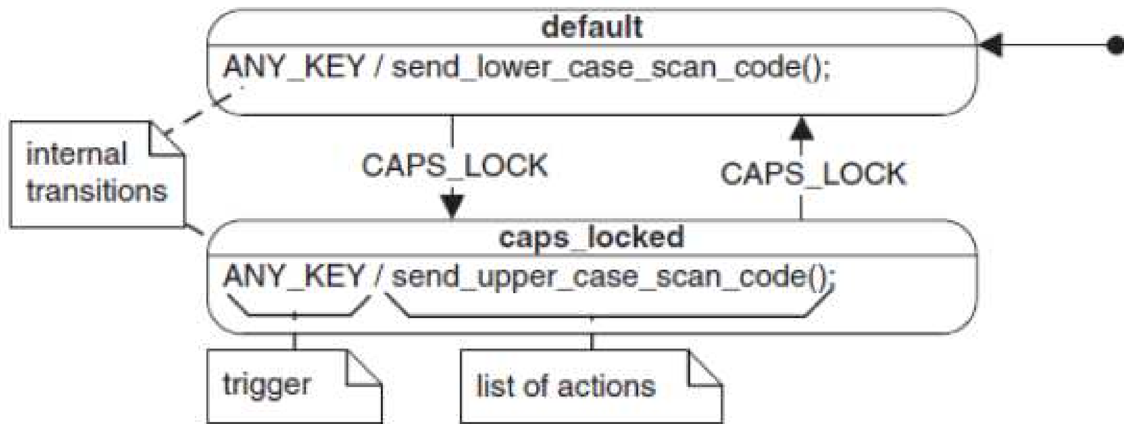

Figure 9: "UML state diagram of the keyboard state machine with internal transitions. (Samek, 2008)"

The third class of actions that needs to be discussed is those associated directly with transitions. These actions are slightly different in UML and the implementation which will be used in chapters 3 and 4. In the UML specification, transition actions are evaluated as follows:

> "*1. Evaluate the guard condition associated with the transition and perform the following steps only if the guard evaluates to TRUE.*
>
> *2. Exit the source state configuration.*
>
> *3. Execute the actions associated with the transition.*
>
> *4. Enter the target state configuration* (Samek, 2008)"

In the QP framework created by Samek, steps 2 and 3 are reversed, with the actions performed after the guard condition is evaluated, but before the source state is exited.

> "*1. Evaluate the guard condition associated with the transition and perform the following steps only if the guard evaluates to TRUE.*
>
> *2. Execute the actions associated with the transition.*

3. Atomically exit the source state configuration and enter the target state configuration (Samek, 2008).

This is both simplifies the programming, as well as recognizes the fact that a state transition needs to be atomic. This need for atomicity is because the state machine is in an uncertain configuration during the transition and thus cannot safely be operated upon. As a result, the author would argue that this departure from the UML specification by the QP framework is an improvement, rather than a flaw.

Orthogonal Regions

Speaking of departures from the UML specification of extensions to state machines, the topic of orthogonal regions bears discussion. There exist objects which seem to be in more than one independent state at the same time. For another keyboard example, take the Num Lock and Caps Lock keys. When typing, the keyboard can be in an on or off state for both keys. Without some manner of being in more than one state at a time, the number of states necessary becomes multiplicative rather than additive. This keyboard example of 2+2=4 or 2*2= 4 may make this difference seem unimportant. Addition of Scroll Lock, for 2+2+2=6 or 2*2*2=8, makes the importance more apparent. It is this problem of independent states that the concept of orthogonal regions was created to address. Orthogonal regions, in the context of state machines, refer to sets of independent states in which a state machine may be simultaneously. To see what the keyboard example looks like, see Figure 10 below.

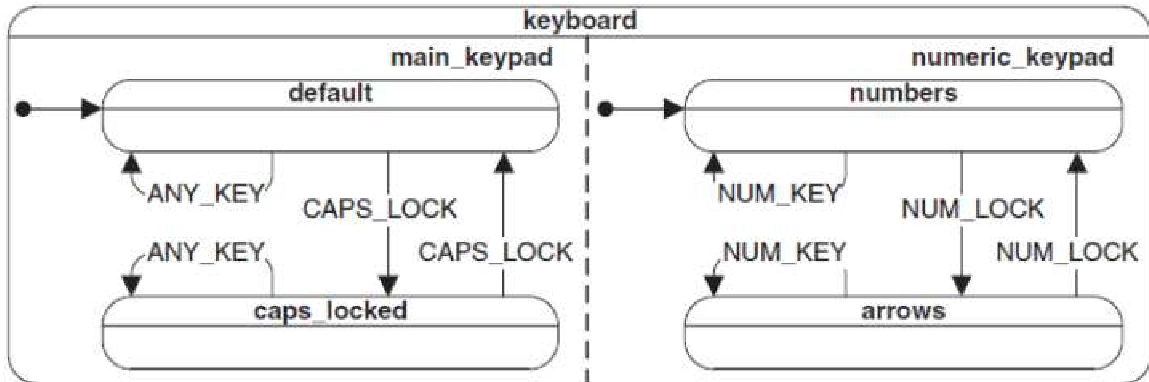

Figure 10: Orthogonal Regions of keyboard (Samek, 2008)

Orthogonal regions are not strictly supported in the QP framework under discussion as they are an expensive feature. This is because "*Each orthogonal region requires a separate state variable (RAM) and some extra effort in dispatching events (CPU cycles)* (Samek, 2008)." Essentially, each region is a separate state that must be separately stored, checked, and processed. This separation necessitates an entirely different coding approach to handling the state machines than the case of single nested states, cluttering up the code and reducing performance. However, the separate nature of the states is also precisely why the lack of direct support for orthogonal regions is not a problem. As pointed out by Rumbaugh et al. and referenced by Samek, orthogonal regions, as a general rule, are produced by aggregation (Rumbaugh, Blaha, Lorenson, Eddy, & Premerlani, 1990). This means that orthogonal regions can be produced by a composition of state machines, rather than a single state machine with orthogonal regions. As an example, take the keyboard: one state for the main keyboard, and one for the numeric keypad. Also another example of aggregation that could be presented as either orthogonal regions or composite objects would be an alarm clock. The state of the clock, and the state of the alarm, either two regions of one object, or a clock containing an alarm object. This object composition is argued by Samek as being superior to the single object-two state case for three main reasons. First is that of code reusability. Creating the regions as separate state machines, one containing the others, they are no longer interdependent and can be reused in other combinations in the future. For example, the

same alarm in another clock, or the numeric keypad on its own. Second, "*The composition of state machines is not limited to just one level. Components can have state machine subcomponents; that is, the components can be containers for lower-level subcomponents. Such a recursion of components can proceed arbitrarily deep* (Samek, 2008)." This is advantageous as it mirrors the reality that many, if not all, real world objects are similarly composed of objects aggregated together. Third, a full implementation of orthogonal regions would imply that all events are dispatched to all regions. A composite implementation, on the contrary, allows the container object to filter out irrelevant events and supply supplemental data. This filtering can obviously drastically cut down on the processing necessary in the case of even moderately complex aggregated systems. For instance, in a state machine model of a keyboard, there is no need for the numeric keyboard component to receive notification of key presses from the main keypad. These advantages, combined with the nature of real world objects and data structures, mean that the lack of direct orthogonal region support by the QP framework is not a flaw. Instead, it is merely a case of the UML specification trying to handle any and all theoretical possibilities, as opposed to being a model for efficient embedded programming.

Computational Limits of Statecharts

So at this point it should be asked: Are there problems that cannot be programmed in the form of a hierarchical, composite, state machine as discussed in this chapter? The answer to this is quite simple. Yes, but no finite, linear-time, digital computer can solve those problems anyway. The reasoning for why these traditional computers have the same limitations is as follows:

1. The circuits in traditional computers are equivalent to finite-state machines, or FSMs (Wright, 2005).

2. Since traditional computers send, receive, process, and store data through such FSM circuits, they are an aggregation of many small FSMs into larger FSMs.

3. Hierarchical state machines (HSMs) of any recursion or composition level are mathematically equivalent to FSMs.
4. Therefore, any traditional computer **is** a HSM.
5. Thus anything a traditional computer is capable of can be modeled as a state machine of the form discussed in this thesis.

As a point of clarification, the emphasis on the notion of a traditional computer is not negligible. There exist certain classes of computers which might very well work on algorithms that cannot be fully represented as HSMs. For an example of such a device, take quantum computers. As the author understands it, the basic principle of quantum computing is to take advantage of the quantum nature of reality. This quantum nature allows very small objects, such as electrons and photons, to be in multiple, even effectively all or infinite, states simultaneously. A state machine on the other hand cannot do this, as it must be in a single composite state at any given time, and cannot respond to events while transitioning between them. Other, more theoretical examples are computers utilizing non-linear time phenomena such as closed time-like curves. Such computers, at least in some theories, rely on either infinite computational time, cause to follow effect, or other exotic concepts. Both violate the finite, cause and effect nature of the state machines discussed. While this means state machines might not be able supply all the needs for some highly experimental and theoretical computational devices, this is not a large limitation. For one, in each case, the quantum or non-linear time component can be seen as merely a source of events that would need to be processed by the more traditional parts of the system. As a result, state machines could very well be useful, if not sufficient, in programing and understanding such computers as well. Of course, just because these state machines *can* be used to program any traditional computer, does not always mean that they will always be the most efficient or practical

solution. A programming and development strategy, if it exists, that was *always* most efficient or practical would certainly be Nobel Prize material.

CHAPTER III

Methodology of State Chart Analysis

As an abstract method of organizing program code, there are many different ways to implement a state chart programming method. There are also, as discussed in Chapter 1, thousands of potential hardware targets to run the code on. For the purpose of this paper, the focus will be on analyzing the use of the QP C++ framework on the Arduino Uno. This approach was chosen for several reasons. For one, unlike most real-time kernels, the QP C++ framework is open source and free to use, as opposed to the thousands of dollars a developer could be expected to need for National Instruments LabView or the venerable VxWorks. Also, the Arduino Uno not only has a compatible development kit, but has several advantages as a hardware platform. For one, the Arduino is cheap, at $40 per board at the authors local RadioShack. For another advantage, the Arduino uses the AVR ATmega328 which uses a stack based program pointer. This pointer method allows for the use of the pre-emptive QK kernel. As one of the world's largest microcontroller manufacturers, AVR is both commonly used and provides a free compiler for use with their chips. This means that for less than $100 and a basic pc, it is possible to have a pair of microcontroller boards to test. As an added bonus, the QP framework comes with free development tools such as the QM modeling tool which allows for graphical programming. Hopefully, the above stated benefits combined with the analysis, methodology, and results that follow will convince the reader that the Arduino and QP framework make a good platform for demonstrating the benefits and performance of the state chart approach to programming.

Timing Characteristics

One of the most important characteristics to understand about any software approach, especially for real-time and/or embedded systems is the timing characteristics of various operations. Given the prime importance of timing the author will attempt to establish a thorough analysis of the timing. In establishing timing characteristics, it is important to show both characteristics of the framework and establish a method by which other target environments can have benchmarks established. With that in mind, it becomes necessary to explore both the method of timing and what to time.

Using the Arduino Uno as a Timer

The attempt to use Windows 7 based programs to reliably time microsecond intervals was unsuccessful. Windows 7 is not a real-time platform and thus lacks the tools for reliably performing such fine measurements precisely. So another approach was devised. Since the Arduino UNO is so cheap, and can be run bare-metal without an overlaying operating system, a second board was purchased to use as a timer. The following is the approach used to establish and test the effectiveness of the Arduino as a microsecond precision timer.

The Arduino library contains a function called pulseIn(). This function is designed to measure pulses between 10 microseconds and 3 minutes in length and return the length of the pulse in microseconds. The function takes as parameters the pin to test, whether the pulse will be high or low, and a timeout value. For simplicity, a high pulse on the first available digital I/O port of 2 was chosen, along with a 10 second timeout. Code running this function on a loop and printing the resulting values was run on one Arduino board.

The second board was then set to send out pulses on a loop. This was accomplished through using a bitwise OR statement on the control register for pin 2 of the second board. The built-in digitalWrite() function was deliberately avoided due to the extra delay it would enter into the

process. The delay length was then established using the built-in delayMicroseconds(). This was followed by another bitwise operation to turn off the pulse. Once programmed, the two boards, running off of the same set of USB ports, had their pin 2 ports connected by a resistor to allow for communication.

By making use of the free TeraTerm software to establish serial communication with the Arduino boards, it was simple to loop through large numbers of timing runs and then copy and paste the results into an excel spreadsheet. Initial runs of this timing method had the results listed in Table 4 below.

Pulse	Mean	Mode	Range	Min	Max
10	8.62	10	15	1	16
15	13.35	14	20	2	22
20	18.07	19	18	7	25
25	22.63	24	19	12	31
30	27.24	29	18	18	36
35	32.2	35	18	23	41

Table 4: Initial Timer Test Results, n=9999

While at first the author was dismayed at the lack of precision and attempted to develop a calibration formula, a set of patterns was noticed. First, was that a linear fit to mean values of the data had a listed R value of .9999 across the collected datasets. This suggested some very consistent source of the errors. Second, it was noticed that aside from the initial pulse of 10us, the range was remarkably consistent at 18-20us. So after a brief search online into the mechanics of the pulseIn() and delayMicrosecond() functions, the author discovered that they work by polling an internal register. The internal register is connected to the oscillator and increments every microsecond.

Since neither function relied on interrupts, the author then attempted turning off interrupts before sending each pulse and re-enabling them after. By doing this on the board measuring the pulse, a whopping 10us was dropped from the range of timer values. By also disabling the interrupts

while sending each pulse, the result was far more precise. By disabling interrupts on both boards, a range in values of 2us was attained, length-2 to length, see Table 5 below.

	Both interrupts enabled	Timer interrupts disabled	Both interrupts disabled
Mean	32.19992	34.6487649	34.4171417
Range	18	8	2
Min	23	33	33
Max	41	41	35

Table 5: Effect of Disabling Interrupts, 35us pulse n=9999

To put these results in perspective, the possible precision needed to be explored. Internally, each function works by polling an internal timer on the chip. Functionally, this means that in each board, the produced or measured time will be between the requested or actual time, and one microsecond smaller. The one microsecond smaller result occurs when polling begins just before the internal timer increments and the accurate time occurs when polling begins just after incrementing. As a result, a 2us board to board precision range is the best possible precision that can be attained with the Arduino boards without resorting to an extensive rewrite using assembly language. Since the Arduino runs at 16MHz, this is a range of 8 computational cycles. While being able to precisely and accurately measure down to the cycle would be ideal, it must be remembered that we are working with less than $100 worth of equipment, and are limited in both our accuracy and precision by the oscillators inside them.

Precision of Timing the QP Framework

With a basic timing system in place, it is now time to explore the timing characteristics of the QP Framework Pre-Emptive kernel on the Arduino Uno. First we must establish a way to interpret the results so we can narrow in on the number of computational cycles needed to perform any given task in the framework, and the uncertainty in our measurements. In order to do that we

need to make a few assumptions about our measurements and what is going on. Our first assumption is the pulseIn() function we are using in the timer board retains its measured and theoretical precision of -1microsecond to +0microsecond during our experiment. Second, we assume that timing differences between the two boards amount to less than one computational cycle. These sources include several sources of error. First is that it is not necessarily true that the test board voltage, rises and falls in a perfectly symmetric fashion. Likewise, the timer may not recognize the change in voltage symmetrically between a rise and fall. Another possible source of error is EMF interference slightly altering the signal voltage along the wire, introducing small changes in pulse symmetry. Quick calculations indicate that light can travel 75m within one clock cycle, so signal propagation delay is likely negligible for the short distance involved. Additionally, the boards' oscillators likely do not operate with exactly the same frequency. Combined, while these timing differences are likely quite small, even a plus or minus of a small fraction of a microsecond can cause, or prevent, an additional timer increment. This is what leads to the addition of a microsecond increase in our measurements uncertainty. Our second assumption is that the processes we are measuring do not need to use cycles in multiples of four. While this seems obvious, it must be remembered that the timer only resolves every four cycles. Thus if a process takes 23 cycles for instance, the timer will see one of two results. The timer will display 20 cycles if the process started less than one cycle after a timer increment, or 24 cycles if the process started less than one cycle before an increment. Next, we assume that the true time the process takes is a constant number of cycles. This assumption follows directly from the combination of the deterministic nature of the framework, and the disabled interrupts during the process. Finally, we assume that when we run our tests, our boards are out of sync by a random factor. This allows us to use statistical analysis of the results. These assumptions and their consequences are summarized below in Table 6: Timing Assumptions and Consequences.

Assumption	Measurement
pulseIn() works	true -1 to +0 microseconds
Max 1 cycle difference	true -1 to +1 cycle
Non-4 possible	true -3 to +3 cycles
Constant time and random sync	Mean approximates true
Total	-2 to +1 microsecond precision
	-8 to +4 cycle precision

Table 6: Timing Assumptions and Consequences

If the assumptions and analysis are accurate, then any timer results will be between 2 microseconds or 8 computational cycles smaller and 1 microsecond or 4 computational cycles larger than the true time a process takes. Further, a large sample size should allow for a mean close to the true value. If any test gives a range of results greater than 3 microseconds, it will be an indication of a flaw in the analysis. Now that we have a means to measure time, we must now decide on what to measure.

State Machine Timing

With a method to time computation processes down, what exactly to be timed must be decided. Three of the most common tasks to be performed by the QP framework are the initialization of state machines into certain states, state changes, and event handling. These tasks are very much the bread and butter operations that underlie the use of the QP framework. Since they will each be used frequently, they are worth timing. Further, a means of describing the exact state combinations possible are needed for clarification.

Much of the operation of the framework in controlling program behavior is in the state transition framework. Transitions between states occur when the state machine needs to change behavior, such as car changing between forward and reverse states. Transitions to the same state essentially act as a reset, exiting and then returning to the same state. Initial transitions occur when the

destination state specifies a substate that is to be entered by default. This is useful in encapsulating and reusing behavior within state machines, as a state transition can allow the destination to decide which if any substates are necessary to finish the transition. Additionally the state machines handle events using the same framework as state transitions. The framework does this through the use of what is called an internal transition. For example, pressing a letter on a keyboard triggers an internal transition to process the key press, but generally does not result in a change in state or reset. Since these initializations and state transitions encompass much of the behavior specific to the use of the state machine framework, they will be the focus of the timing analysis.

Now that what to time has been decided, it is necessary to specify a means of describing the exact behavior being timed. First let us establish a systematic naming system for states such that the relationship between any two states is apparent in the name. The method arrived at by the author is to specify the name of any given state by the letter "s" followed by a string of numbers. The "s" simply specifies that this is a state. The number is where the location of the state is encoded. In order for this encoding to make sense, one must understand the nesting levels of a state. As an analogy to help in understanding nesting level, and the naming convention, think of a state as a box. A top level state is one which is resting directly on a table. Inside this box might be smaller boxes, which can have boxes inside as well, repeating with smaller and smaller boxes. The nesting level of a state is how many boxes must be opened in order to look inside. To look inside a top level state, one needs only to open the state itself, or nesting level 1. Nesting level 2 states would be represented as boxes within another box resting on the table. This process can continue, like Russian nesting dolls up to the nesting level limit in the given configuration of the QP framework, 6 by default. Back to our analogy, what if we have two boxes on the table, three, or more. To name them all the same would confuse things, as each state needs a unique name. So we indicate which one is which by naming the first s1, the second s2, and so on. We do this from

left to right, top down, as we read our state diagram. We can then easily picture which one goes where. For example see Figure 12: Naming Digit value vs. Location.

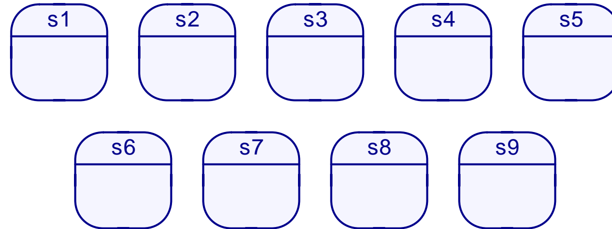

Figure 11: Naming Digit value vs. Location

Now let us picture a different scenario. In this scenario, we have 6 boxes, one within the other up to our nesting limit. We cannot simply call these s1, s2, s3, etc. as we would then be unsure if our boxes were nested, or merely side by side. So what we do instead is add an extra digit at each nesting level. For our analogy, this means that each box has one more digit than the box it is in, as in Figure 12: Nesting Level Digits.

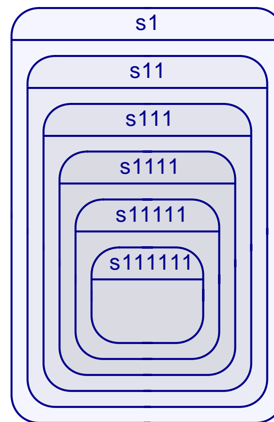

Figure 12: Nesting Level Digits

Now, here comes the tricky part, we can have several boxes on the table, each with its own name. We can also have boxes nested within each other. However, what happens if we have a little of both? Well if we follow a few simple rules, it will not be a problem. First we name our top level boxes s1, s2, s3, and so on as above. Then, whenever we put one box inside of another, we

simply name the smaller box after the bigger. Instead of being "s" plus 1, 2, 3, etc. from left to right, we replace "s" with the name of the box we are adding to. So if we add two boxes to state s2, we would call the first one on the left s21 and the second on the right s22. The first box added to s22 would be s221. As an example, see Figure 13: Full Naming Convention Example. This allows us to uniquely identify any state.

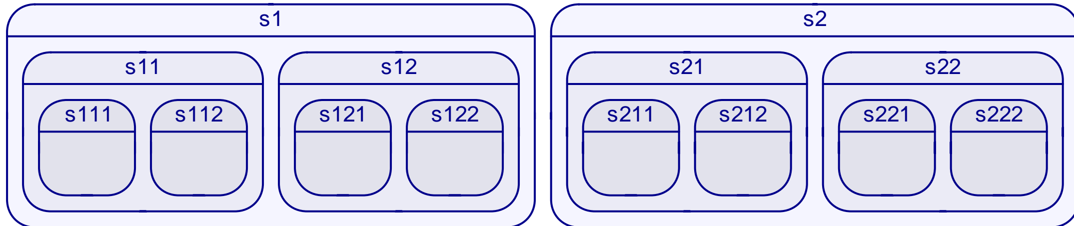

Figure 13: Full Naming Convention Example

This convention also allows us to quickly deduce which states/boxes a state is contained within. Simply remove the last digit from the name of a state, and you then have the name of the superstate. Repeat as necessary. The only limit to this convention is that it is limited to 9 substates of any given state, including the top level. Later, it will be shown that a limit of 9 does not prevent an analysis of relevant behavior. However, one could us 0 as a digit, raising the limit to 10. Or the hexadecimal a, b, c, d, e, and f could be used to give 16. If hexadecimal is insufficient for the user's purpose, the addition of a separator character such as the underscore would remove the limit entirely.

Now that we have a means of unambiguously naming states, we need to review what kinds of relationships states can have to each other. The most basic relationship two states can have to each other is to be superstate and substate, such as s1 and s11. This relationship can be deduced quickly from the names. The substate s11 contains the name of its parent superstate s1, with one additional digit. In addition to being direct superstate and substate, it is possible for a state to be nested multiple levels below a given superstate, such as s1112 and s1. In this multilevel nesting case, state s1112 is the second substate of s111, which is the first substate of s11, which is the

first substate of s1. A given state may have multiple substates within itself, such as s1 having s11 and s12 as substates. This relationship, having the same superstate, makes states s11 and s12 what the author calls parallel substates. Put in other words, parallel substates are related through a sibling pair, rather than only parent-children pairs. States s1, s2, s3, etc. do share the same hidden top level superstate, the table. This hidden superstate is, however, treated differently in the code. As such top level states like s1 and s2 are not considered parallel substates for our discussion, although they are in fact parallel. The naming scheme makes for easy identification of parallel substates. Any state with a digit, beyond the first, that is higher than 1 is a parallel substate, or contained within one. With a means of quickly describing the relationship between states, we can move on to describing transitions.

The concept of transitions is important for the function of state machines. The exact functioning of transitions is also critical for timing of the framework. The simplest type of transition we can have is what is called an internal transition. An internal transition performs some action, possibly with a guard condition, but does not result in a change in state. This is essentially the means by which events which do not require a change in state are processed by the state machine. Another type of transition vital for flexible specification of state machines is the initial transition. As discussed previously, an initial transition in a state specifies that the state machine should enter a specific substate. Taking these concepts into account leads to what the author calls a basic transition. A basic transition is a transition which results in a change in state, but neither needs to distinguish between parallel substates (siblings) nor triggers an initial transition. These clarifications prove useful in defining exactly what needs to be timed.

Each transition has an origin and a destination. The origin of a transition is defined by both the state the transition is associated with and the state which the state machine is in at the time. This distinction is necessary due to the way superstates and substates behave. For example, a transition may be defined in state s1 to occur in response to event A. If the state machine is in

state s11 which does not define a response to event A, the transition from s1 will be triggered as a result of Event A. This is distinct from the case where the state machine is in state s1, as there is no need to check if s11 defines a response. As a result, one must specify both the state the transition is associated with, and the state which the transition is being called from in order to fully define the origin of a transition. For example a state machine with s1 and s11 will have 3 possible origins: s1 while in s1, s11 while in s11, and s1 while in s11. The next piece of information needed to unambiguously identify a transition is the destination. The destination is defined by the destination state, and the initial transition(s), if any, which are triggered as a result. For instance, if s1 is specified as the destination, it matters whether s1 has an initial transition specified to s11, s111, etc. Likewise specifying s11 is distinct from specifying s1 with an IT (initial transition) to s11, as extra calculations are necessary. Thus we specify if an initial transition or transitions take place. For example s1, s1 - IT to s11, or s11 are three possible destinations. Now we have a means of specifying the parameters of both the origins and destinations of possible transitions.

At first glance, it may appear as if there is an effectively infinite number of combinations of origins and destinations for transitions, since there is a nearly unlimited number of possible states. However, we can narrow things down. With a little understanding of the way the framework deals with these transitions we can combine the infinite possibilities into a finite number of possible unique configurations. One of the biggest means of narrowing things down is the behavior of parallel substates. To understand this behavior, we need to understand how states are linked. The algorithm which determines the path of states to enter and exit during a transition starts with only two parameters, origin state and target state. In the QP framework, a state contains a reference to its direct superstate. This superstate reference is the ***only*** means by which the transition algorithm can determine the relationship between states. In other words, the algorithm can only ask a state who its parent is, not its children or siblings. The algorithm uses

these three pieces of information available to search for the deepest common superstate of origin state and destination state. For instance s11 is the common superstate of s111 and s1121, while s1 is the common superstate of s1 and s11. This behavior has a valuable consequence: A state knows only itself, its superstate, and the destination of its transitions. As a result, the existence and number of any substates or parallel substates not in the direct path is completely irrelevant. Only the current state of the state-machine, the state the transition is defined in, the destination, the least common superstate, and any traversed states matter. Or, put another way a transition can move only up a nesting level 0 or more times, sideways 0 or 1 times, then down a nesting level 0 or more times, in that order. Brief testing of various combinations bears this out. This allows us to prune the infinite possibilities down a great deal.

With constraints on the infinite possibilities we can now construct what shall be called prototype state diagrams. A prototype state diagram has the minimum number of states that allow for exploration of every possible transition. One prototype exists for each nesting level. A prototype state machine can be constructed by the use of a few simple rules. First, since parallel substates not in the path do not matter, there only needs to be two substates per state. This is because, at most, only two substates in the same superstate can possibly be in a single path from origin to destination. Second, a pair of substates with the same superstate need only appear once per nesting level. This was done in the leftmost (lowest digit value) state for simplicity. Third, each state should have at least one substate up to the nesting level limit. These rules result in the following prototype state machines for nesting levels (NL) one through four below in Figure 14: Prototype State Diagrams.

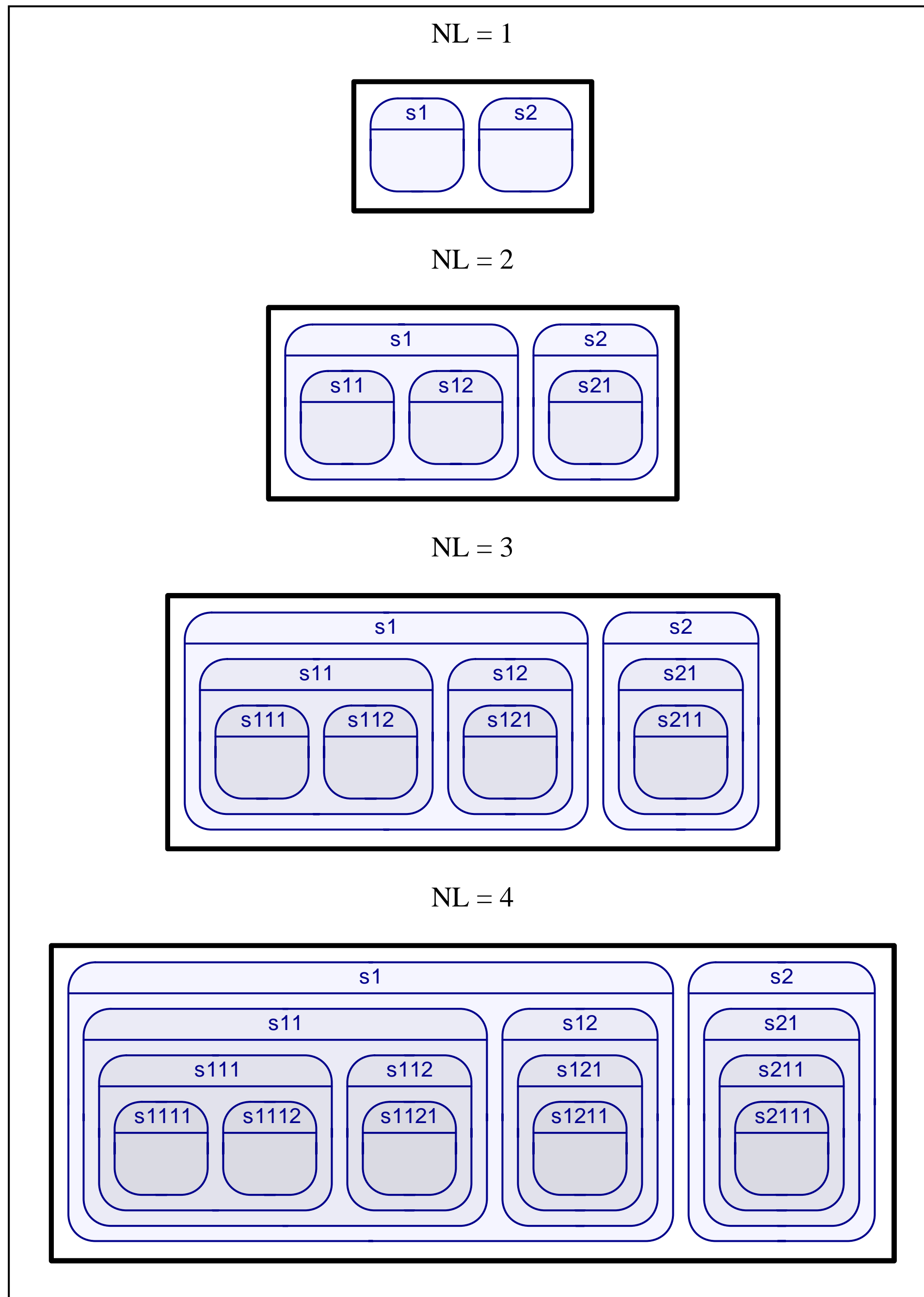

Figure 12: Prototype State Diagrams

A first attempt to list all of the combinations of transitions proved haphazard and frustrating. A large part of this was uncertainty as to which combinations, if any, had yet to be tried. After a little reflection, the author decided it would be best to find a way to quantify the combinations

needed. The development of the naming scheme, state relationships, and the prototype state diagrams were integral to this effort. However, there was still the need to determine the number of transitions to measure.

In order to quantify the number of transitions possible at any given nesting level, mathematical models had to be developed. This was done by printing off the prototype state diagrams and then trying to exhaustively list every possible origin and destination. For reference, these list are given in Appendix C: Lists of Origins and Destinations. These were recorded on an excel spreadsheet starting with nesting level 1 then moving on to nesting level 2, then 3, then 4. These origins and destinations were then carefully scrutinized for any patterns that could lead to an equation. Once these equations had been found for the first three nesting levels, adding missing destinations as they were discovered, the author noticed that no new patterns seemed to appear at nesting level 4. Thus the fourth nesting level was used as a check due to the inductive nature of the equation development.

The first, and simplest, of these equations specified the possible origins for a transition. Because only the relationship between origin and destination matters for timing purposes, this number is rather small. For example, a transition between s1 and s2 takes the same time as one between s2 and s1. This allowed for the origin states to be constrained to s1, s11, s111, etc. A transition may also be called from a substate of the state it is associated with. This adds a number of origins equal to the nesting level each time a new nesting level is added. Taking these factors into account leads to the following Formula 1: Unique Origins

$$\sum_{n=1}^{NL} n$$

Formula 1: Unique Origins

Developing an equation for possible destinations proved much more difficult. However, breaking the task into pieces produced better results. First, the transition might be an internal transition,

resulting in one transition per origin. Second, basic transitions are relatively easy to derive from the prototype state diagrams. Combining these two relatively simple transitions gives us the following Formula 2: Basic Destinations.

$$2NL + 1$$

Formula 2: Basic Destinations

The development of an equation modeling the number of possible initial transition combinations proved quite difficult at first. Originally the author had been searching for polynomial models and painstakingly writing down every found combination. Eventually it was noticed that some of the patterns, as well as the original polynomial form of Formula 1 could be represented as summation equations. This epiphany, combined with the concept of parallel substates, was the missing key. Ignoring parallel substates for the moment, the number of unique destination paths with initial transitions can be modeled by Formula 3: Non-Parallel Initial Transition Combinations.

$$2\sum_{a=2}^{NL}(2^{a-1} - 1)$$

Formula 3: Non-Parallel Initial Transition Combinations

The combination of Formula 2 and 3 cover all the destination paths that do not include parallel substates. Next, there is the need to determine the number of such states. This resulted in Formula 4: Number of Parallel Substates.

$$\sum_{a=2}^{NL}(a - 1)$$

Formula 4: Number of Parallel Substates

The final form of transition destinations include those in which an initial transition or transitions occur in parallel substates. While the most detailed formula thus far, the patterns were the same, leading to Formula 5: Parallel Initial Transition Combinations.

$$2\sum_{a=3}^{NL}\sum_{b=a}^{NL}(2^{b-a+1}-1)$$

Formula 5: Parallel Initial Transition Combinations

A careful look at the formulae shows a few things noticed in the original patterns. First, initial transitions do not begin to occur until nesting level 2. This makes sense as an initial transition requires the existence of substates. Parallel substates begin at nesting level 2 as well for the same reason. At nesting level 3, parallel substates can support an initial transition as well, leading to Formula 5. At nesting levels of four or more, no additional types of behavior seem to occur. This allows us to formulate the total number of possible unique transitions. Adding up our results gives the following Formula 6: Total Unique Transition Destinations and Formula 7: Total Unique Transitions.

$$2\sum_{a=3}^{NL}\sum_{b=a}^{NL}(2^{b-a+1}-1)+2\sum_{a=2}^{NL}(2^{a-1})+\sum_{a=2}^{NL}(a-1)+2NL+1$$

Formula 6: Total Unique Transition Destinations

$$Transitions = Origins * Destinations$$

Formula 7: Total Unique Transitions

Calculating these values for each nesting level up to the default limit of 6 gives Table 7: Origins, Destinations, and Transitions per Nesting Level.

NL	(1) Origins	(2) Basic Destinations	(3) Non-Parallel ITs	(4) Parallel substates	(5) Parallel ITs	(6) Total Destinations	(7) Total Transitions
1	1	3	0	0	0	3	**3**
2	3	5	2	1	0	8	**24**
3	6	7	8	3	1	19	**114**
4	10	9	22	6	5	42	**420**
5	15	11	52	10	16	89	**1335**
6	21	13	114	15	42	184	**3864**

Table 7: Origins, Destinations, and Transitions per Nesting Level

Now that we know what exactly is possible at any given nesting level we can begin the process of actually gathering the results. In order to do this efficiently, we need a flexible and easily adjustable means of testing the various combinations. Thanks to the graphical programming interface included with the free modeling tool called QM, this proved relatively simple. QM is designed as a means of graphically programming state machines in the QP framework. For an example, see Figure 13: QM State-Machine Example below.

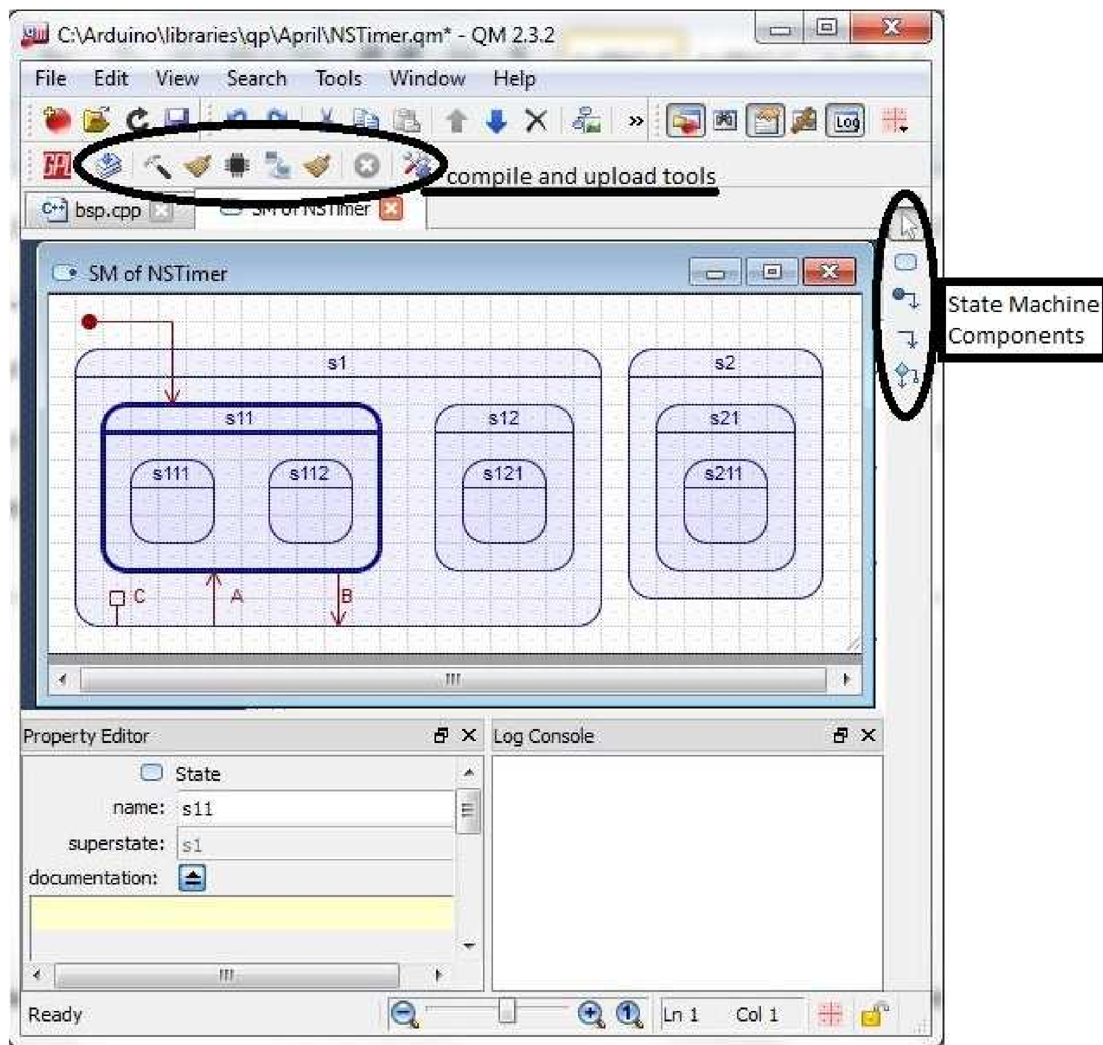

Figure 13: QM State-Machine Example

Now on to what is being timed. An early attempt would start a pulse, start the transition to be timed, and then stop the pulse in the entry action of the destination state. This approach was

unsuccessful because an entry action only occurs when the state is entered from outside. Going from state s11 to state s1 would not trigger an entry. This problem, along with the need to return to the origin state before another transition could be timed required a different approach. After a few iterations and refinements, the author arrived at the following code in bsp.cpp (board support package C++ file).

```
void QF::onIdle() {
        //We are idle, so we are done processing
        PULSE_OFF();
        //Flip flag on and off for test/reset
        flag!=1;
        //test if flag off
        if(!flag)
        {
                //Make sure timer board has time to prepare
                delay(15);
                //Start Pulse
                PULSE_ON();
                //Post event signal A to start transition
                AO_NSTimer->POST(Q_NEW(QEvt, A_SIG), &onIdle);
        }
        //Return to Origin if flag is on with signal B
        else
        {
                AO_NSTimer->POST(Q_NEW(QEvt, B_SIG), &onIdle);
        }
}
```

Figure 14: Pulse Timing code in QF::onIdle()

The code above in Figure 14: Pulse Timing code in QF::onIdle() creates a pulse starting just before the creation of the event A which triggers the transition to be timed. The pulse then ends as soon as the idle function is called. This allows us to measure the time it takes to fully process a transition from start to finish, including the time to create and dispose of the event instance. The

code then sends signal B which resets the state machine so another round can begin. With this code, we need only follow the following steps to time a transition:

1. Draw the state machine
    a. Draw an initial transition to the origin state.
    b. Draw Signal A to represent the transition destination
    c. Draw any initial transitions that are part of the destination.
    d. Draw signal B to return from ending state back to starting state.
    e. Use signal C to ensure that origin state has two signals visible for consistent timing.
        i. Each state uses switch case statements to process signals. As such the number of signals available has a small (~1us) effect.
2. Press the Make (hammer) button in the QM compiling tools to compile program
3. Press the upload (chip) button to upload to Arduino.
4. Press reset button on timer Arduino.
5. Wait for timer board to collect and print 999 time samples.
6. Copy results from TeraTerm Serial monitor to excel sheet.
7. Record Min, Mean, and Max times
8. Repeat.

With this set of steps, the author proceeded to collect the min, max and mean measurements for every possible transition for the first three nesting levels of the QP vanilla cooperative kernel. The results are presented and discussed below in Chapter 4.

CHAPTER IV

Timing Results and Discussion

Nesting Level 1

That timing results have been gathered for every possible transition for the first three levels. It is now time to present and discuss them, starting with Table 8: Timing Results for Nesting Level 1.

Min (us)		Transition Origins
		s1(s1)
Transition Destinations	s1	56
	s2	45
	Internal	44

Mean (us)		Transition Origins
		s1(s1)
Transition Destinations	s1	56.854
	s2	46.021
	Internal	44.496

Max (us)		Transition Origins
		s1(s1)
Transition Destinations	s1	58
	s2	48
	Internal	45

Floor (cycles)		Transition Origins
		s1(s1)
Transition Destinations	s1	224
	s2	180
	Internal	176

Mean (cycles)		Transition Origins
		s1(s1)
Transition Destinations	s1	227.42
	s2	184.08
	Internal	177.98

Ceiling (cycles)		Transition Origins
		s1(s1)
Transition Destinations	s1	232
	s2	192
	Internal	180

Table 8: Timing Results for Nesting Level 1

Nesting Level 1 does not give much to discuss. The state before the parenthesis is the state in which the transition is defined while the state in parenthesis is the current state of the state machine. Thus s1(s11) would be read as the origin “from s1 while in s11.” The maximum range in the results was 3us for the second transition. This matches well with the theoretical discussion of the timing characteristics in Chapter 3 page 42. There are only 3 possible transitions for this nesting level. The first represents a state transition exiting and leaving the same state. The second represents a transition between two states. The surprise here is that it takes approximately 24% longer to execute a transition to self than to another state. It is possible that this is due to the order the transition code searches for the path to take. The third data point is for an internal transition. That an internal transition is the quickest is no surprise. The only curiosity is that the state-to-state transition took only 3.4% longer than the internal transition. One calculation that helps to put the speed of the framework into focus is how many transitions could be completed per second. This works out to approximately 17500, 21700, or 22400 transitions per second on the ATmega328. Not bad for a cheap 16Mhz chip. Of course if a program actually did that many operations it would have no time left for anything else. On a more practical note, 1000 random transitions would use up 4.9% of the clock cycles as overhead on average. While the author lacks comparison data points for other operating systems and kernels, ~5% CPU overhead for 1000 operations per second seems rather effective. Given the ease of organizing program behavior through the combination of state machines in the QP framework and QM modeling tool for drawing the state charts, this seems rather attractive to the author. We will of course need to see how this pattern holds at higher nesting levels.

Nesting Level 2

Min (us)		Transition Origins		
		s1(s1)	s11(s11)	s1(s11)
Transition Destinations	s1	56	45	56
	s1 - IT to s11	73	70	69
	s11	61	56	57
	s12	61	48	61
	s2	45	45	45
	s2 - IT to s21	48	48	48
	s21	48	48	48
	Internal	44	44	45

Mean (us)		Transition Origins		
		s1(s1)	s11(s11)	s1(s11)
Transition Destinations	s1	56.872	46.013	56.875
	s1 - IT to s11	73.141	70.086	69.036
	s11	61.969	56.856	57.881
	s12	61.977	48.349	61.970
	s2	46.015	45.996	46.011
	s2 - IT to s21	49.010	49.000	48.999
	s21	48.377	48.355	48.360
	Internal	44.512	44.481	45.931

Max (us)		Transition Origins		
		s1(s1)	s11(s11)	s1(s11)
Transition Destinations	s1	58	48	57
	s1 - IT to s11	74	71	70
	s11	64	57	60
	s12	64	49	64
	s2	48	48	48
	s2 - IT to s21	50	50	50
	s21	49	49	49
	Internal	45	45	48

Floor (cycles)		Transition Origins		
		s1(s1)	s11(s11)	s1(s11)
Transition Destinations	s1	224	180	224
	s1 - IT to s11	292	280	276
	s11	244	224	228
	s12	244	192	244
	s2	180	180	180
	s2 - IT to s21	192	192	192
	s21	192	192	192
	Internal	176	176	180

Mean (cycles)		Transition Origins		
		s1(s1)	s11(s11)	s1(s11)
Transition Destinations	s1	227.49	184.05	227.50
	s1 - IT to s11	292.56	280.34	276.14
	s11	247.88	227.42	231.52
	s12	247.91	193.40	247.88
	s2	184.06	183.98	184.04
	s2 - IT to s21	196.04	196.00	196.00
	s21	193.51	193.42	193.44
	Internal	178.05	177.92	183.72

Ceiling (cycles)		Transition Origins		
		s1(s1)	s11(s11)	s1(s11)
Transition Destinations	s1	232	192	228
	s1 - IT to s11	296	284	280
	s11	256	228	240
	s12	256	196	256
	s2	192	192	192
	s2 - IT to s21	200	200	200
	s21	196	196	196
	Internal	180	180	192

Table 9: Timing Results for Nesting Level 2

Moving on to the next nesting level, we have Table 9: Timing Results for Nesting Level 2. One of the first patterns to notice is that the transitions we timed in nesting level 1 have not changed more than ~1/10th of a cycle. This strongly suggests that the true time has remained unchanged. In other words, substates beneath both origin and destination do not affect timing. Another easily discernible pattern is that internal transitions take the same amount of time to execute as long as the transition is defined in the current state. Calling an internal transition of a superstate applies an approximately 1.5us (6 cycle) overhead to the transition.

A surprising result is that the time it takes to transition to any of the s2 destinations is completely independent of the origin in s1. In other words if the common superstate of origin and destination is the hidden top level superstate, the origin does not affect the timing. Additionally the s2 transitions were among the fastest. It is likely that this is a result of the order in which the transition algorithm searches for the necessary path. These s2 transitions also allow us to discern that an initial transition to a direct substate takes 3us or 12 cycles, while selecting a destination one nesting level lower takes ~2.3us or 9 cycles extra. It remains to be seen whether this pattern holds. Another result is that the s11 and s12 destinations take equal time from s1(s1) (again read as from s1 while in s1). To put this in more accessible terms, the time it takes to transition into a direct substate from a superstate is independent of which substate is the destination. Of note here is that the algorithm appears to check for destinations in substates after superstates or parallel states. The transitions with s1 as the destination present an interesting pattern. The s1(s11) transition takes the same time as the s1(s1) transition. The final and somewhat confusing pattern in the data is that the destination s1 – IT to s11 takes the longest time to occur. What makes this confusing is that the transition s2 – IT to s21 took only an additional 12 cycles for each origin, while the s1 versions take 65, 96, and 98 extra cycles. The difference between the 96 and 98 could be related to the switch case signal processing, but 65 is too different to result from that. The ~30 cycle discrepancy could be related to the lack of need to exit s11 before the initial transition, but there is probably something else going on as well. Calculating our 1000 operation overhead for this nesting level gives us: max 7.3%, min 4.45%, and 5.4% for even distribution.

Nesting Level 3

Now moving on to the third nesting level, the mean time and mean cycles is shown below in Tables 10 and 11, with the rest shown in Appendix B: Data Tables. Some of our previous patterns hold, while some do not. First, we get the same results as in nesting level 2. Second, the s2 destinations are again independent of the origin and among the fastest. This strongly suggests

that the algorithm is somewhat optimized for transitions between top level states, s1, s2, etc. The 12 and 9 cycle cost for initial transitions or one nesting level for destination noted in the second nesting level do not appear to be a constant. However all s2 destinations take 192 to 212 cycles, a range of only 20 cycles or 5us suggesting a relatively small cost.

	Mean (us)	Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	56.900	46.007	46.012	46.009	56.977	56.959
	s1 - IT to s11	73.131	70.082	49.005	70.197	73.125	69.036
	s1 - IT to s111	81.190	81.455	82.348	77.352	77.075	81.184
	s1 - IT to s11 - IT to s111	87.905	86.763	88.799	82.677	84.636	87.908
	s11	61.966	56.873	48.378	56.956	61.957	57.863
	s11 - IT to s111	77.651	73.135	70.077	69.030	73.548	77.655
	s111	75.026	61.985	56.879	57.863	70.976	75.036
	s112	75.033	61.970	49.757	61.967	75.028	75.032
	s12	61.965	48.363	48.349	48.319	61.970	61.963
	s12 - IT to s121	77.684	50.737	50.703	50.667	77.635	77.658
	s121	75.032	49.778	49.781	49.750	75.026	75.023
	s2	46.005	46.015	46.010	46.006	46.011	46.001
	s2 - IT to s21	49.002	49.006	49.004	49.003	49.003	48.998
	s2 - IT to s21 - IT to s211	51.993	52.000	51.996	51.997	51.997	51.999
	s2 - IT to s211	50.719	50.693	50.717	50.683	50.667	50.719
	s21 - IT to s211	50.621	50.689	50.725	50.691	50.693	50.667
	s21	48.354	48.356	48.361	48.342	48.346	48.320
	s211	49.756	49.790	49.766	49.769	49.775	49.764
	Internal	44.510	44.515	44.513	45.917	48.243	45.923

Table 10: Mean transition time for Nesting Level 3

Mean (cycles)		Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	227.60	184.03	184.05	184.04	227.91	227.84
	s1 - IT to s11	292.52	280.33	196.02	280.79	292.50	276.14
	s1 - IT to s111	324.76	325.82	329.39	309.41	308.30	324.74
	s1 - IT to s11 - IT to s111	351.62	347.05	355.20	330.71	338.54	351.63
	s11	247.86	227.49	193.51	227.82	247.83	231.45
	s11 - IT to s111	310.60	292.54	280.31	276.12	294.19	310.62
	s111	300.10	247.94	227.52	231.45	283.90	300.14
	s112	300.13	247.88	199.03	247.87	300.11	300.13
	s12	247.86	193.45	193.40	193.28	247.88	247.85
	s12 - IT to s121	310.74	202.95	202.81	202.67	310.54	310.63
	s121	300.13	199.11	199.12	199.00	300.10	300.09
	s2	184.02	184.06	184.04	184.02	184.04	184.00
	s2 - IT to s21	196.01	196.02	196.02	196.01	196.01	195.99
	s2 - IT to s21 - IT to s211	207.97	208.00	207.98	207.99	207.99	208.00
	s2 - IT to s211	202.88	202.77	202.87	202.73	202.67	202.88
	s21 - IT to s211	202.48	202.76	202.90	202.76	202.77	202.67
	s21	193.42	193.42	193.44	193.37	193.38	193.28
	s211	199.02	199.16	199.06	199.08	199.10	199.06
	Internal	178.04	178.06	178.05	183.67	192.97	183.69

Table 11: Mean Transition cycles for Nesting Level 3

The s12 destinations (s12, s12 - IT to s121, and s121) show a similar result to the s2 destinations. When transitioning from any origin from within s11, the state parallel to s12, the transition time is independent of origin. Additionally, the only way to get into s12, s121, etc. faster is from a substate. This again suggests that the algorithm is optimized for transitioning between parallel states. This parallel state optimization is shown again when transitioning between s111 and s112. As it was in Nesting Level 2, the time it takes to transition into a direct substate from a superstate is independent of which substate is the destination. At Nesting Level 3 we can also see that traveling from a state to a substate nested 2 levels below is independent of substate as well, although more time consuming than traveling one nesting level. Additionally, this holds true for entering a direct substate then undergoing an initial transition into the next nesting level. This strongly suggests that, for a transition defined above the states listed in the destination, only the combination of nesting levels descended and initial transitions matter. As in previous nesting levels, the time to process internal transitions is short and depends only on the degree of separation between the origin states. The time it takes for transition destination s1 seems to

depend only on whether the transition is defined in s1 or a substate of s1. The non-internal transitions from origins s1(s11) and s1(s1) match as long the transition does not end up in s11. The non-internal transitions from origins s11(s11) and s11(s111) match as long the does not end up in s111. In both cases, when the transition ends up in the substate, the s1(s11) or s11(s111) transitions are slightly faster. These two patterns suggest that the framework is designed such that passing an event up one state does not take additional time. Additionally, the framework seems to remember the state from which passed from, saving time if the destination takes it back. While there are likely more patterns in the data, possibly some which are only noticeable at higher nesting levels, there are a few things to keep in mind. First is that since the number of signals in a state has a small effect on the timing due to the switch case in the signal processing. This might be partially distorting deeper patterns in the data and would certainly cause minor deviations from the patterns in practice. Second is that, as shown in Table 12: Reduced Dataset for Nesting Levels 1-3, we can use these patterns to significantly reduce the number of data points required to characterize the timing behavior. At this point we only need 41 data points to describe the first 3 nesting levels, a reduction of 70.9% from the theoretical 141. Last, back to our practical note, it would cause between 4.45% and 8.88% overhead for 1000 transitions per second with an average of 5.9%.

<table>
<tr><td>Mean (cycles)</td><td colspan="6">Transition Origins</td></tr>
<tr><td>No Initial Transitions</td><td>s1(s1)</td><td>s1(s111)</td><td>s1(s11)</td><td>s11(s11)</td><td>s11(s111)</td><td>s111(s111)</td></tr>
<tr><td>s1</td><td colspan="3">228</td><td colspan="3">184</td></tr>
<tr><td>s11</td><td colspan="2" rowspan="2">248</td><td>231</td><td>227</td><td>228</td><td>194</td></tr>
<tr><td>s12</td><td></td><td colspan="3">193</td></tr>
<tr><td>s111</td><td></td><td>284</td><td></td><td></td><td>231</td><td>228</td></tr>
<tr><td>s112</td><td colspan="3" rowspan="2">300</td><td colspan="2">248</td><td></td></tr>
<tr><td>s121</td><td colspan="3">199</td></tr>
</table>

<table>
<tr><td>Initial Transition Chains</td><td>s1(s1)</td><td>s1(s11)</td><td>s1(s111)</td><td>s11(s11)</td><td>s11(s111)</td><td>s111(s111)</td></tr>
<tr><td>s1 - IT to s11</td><td>293</td><td>276</td><td>293</td><td>280</td><td>281</td><td>196</td></tr>
<tr><td>s1 - IT to s111</td><td colspan="2">325</td><td>308</td><td>326</td><td>309</td><td>329</td></tr>
<tr><td>s1 - IT to s11 - IT to s111</td><td colspan="2">352</td><td>339</td><td>347</td><td>331</td><td>355</td></tr>
<tr><td>s11 - IT to s111</td><td colspan="2">311</td><td>294</td><td>293</td><td>276</td><td>280</td></tr>
<tr><td>s12 - IT to s121</td><td colspan="3">311</td><td colspan="3">203</td></tr>
</table>

<table>
<tr><td>s2 Destinations</td><td>Independent of Origin</td></tr>
<tr><td>s2</td><td>184</td></tr>
<tr><td>s2 - IT to s21</td><td>196</td></tr>
<tr><td>s2 - IT to s21 - IT to s211</td><td>208</td></tr>
<tr><td>s2 - IT to s211</td><td rowspan="2">203</td></tr>
<tr><td>s21 - IT to s211</td></tr>
<tr><td>s21</td><td>193</td></tr>
<tr><td>s211</td><td>199</td></tr>
</table>

<table>
<tr><td>Internal Destinations</td><td>s1(s1)</td><td>s11(s11)</td><td>s111(s111)</td><td>s11(s111)</td><td>s1(s11)</td><td>s1(s111)</td></tr>
<tr><td>Self</td><td colspan="3">178</td><td colspan="2">184</td><td>193</td></tr>
</table>

Table 12: Reduced Dataset for Nesting Levels 1-3

Overall, the results of timing the various transitions for the first three nesting levels shows that to QP framework can process any transition within 3 nesting levels at between 178 cycles or 44.5us and 355 cycles or 88.75us. It also shows that the framework is somewhat focused on performing internal transitions and transitions between parallel states quickly. Since these are likely among the most common transitions a state machine will undergo during operation in practice, this is likely near ideal. Now with the data gathered and reviewed, it is time to get back to our evaluation of the QP state machine framework.

CHAPTER V

Conclusion and Recommendations

Conclusions

In the beginning of this thesis, the author identified several characteristics that a software programming strategy needed in order to address the needs of modern developers. Now that the QP state machine platform has been discussed in some detail, it can be evaluated with these characteristics in mind. The authors objective is to convince the reader that this approach is worthy of consideration. One of the most vital characteristics in showing the worth of the approach taken with QP is broad applicability.

A programming strategy should, perhaps first of all, be applicable to a wide range of problems. This is an area where QP excels. This is because the QP platform is founded upon a lightweight implementation of hierarchical state machines (HSM). As discussed in Chapter 2 pages 34 and 35, any traditional computer is at its heart a HSM. As a result any code that could be stored and run on such a computer could be written as a HSM. One of the main limitations built in to the QP framework is that a maximum of 63 Active Objects can be active in a program. This limit is not as bad as it might seem. It only means that there cannot be more than 63 active threads of execution on a given chip. It is conceivable for the QP framework to run a major server, the power grid of a large city, or something small enough to put inside a dollar store toy. This variety of applications requires flexibility in hardware choices.

There are some programming methods, quite easy to learn and quite flexible, who have the major limitation of hardware choice and portability. Linux, LabView, Windows, and Mac OS, are software platforms with large user and developer bases. These platforms, however, require powerful and expensive microprocessors to run. They are simply unsuitable for something like the pressure sensors in a car tire, or a thermostat. The QP platform, on the other-hand comes in three versions, QP-C, QP-C++, and QP-nano, with even the heaviest, QP-C++, running easily on the Arduino UNO's microcontroller. Additionally, much of the behavior of the code is defined in the state machines and platform. Only the board support package (bsp) needs to change with new hardware. This allows for highly portable code, insensitive to hardware changes, which is a vital requirement for the embedded system developer.

Related to the issue of hardware choice is the issue of memory use. If a program uses extensive memory, it requires more hardware resources. A detailed analysis of the memory usage of the QP-C++ platform was not the focus of this paper. As such its exact parameters are unknown. It can be said one of the nesting level 3 programs used 7278 bytes of flash memory to program. The stack usage is unknown. This is an area rich for future exploration. However, as this was the heavy version of the platform with 9 states and 3 signals using less than 8kb of program memory, it would seem to be relatively lightweight.

A benefit of the relatively lightweight nature of the code, combined with the QM modeling tool, is refinement speed. Once the board support package for a platform is put together, much of the development time is spent on altering the state machines. As the author knows from experience gathering the data for Chapter 4, QP allows for rapid change. Once up and running, it could easily take less than 30 seconds to rearrange a few of the state machines, recompile, and upload using the QM tool. This means that code refinement and implementation can proceed quite quickly. During the development of a product, the ability to add and alter features as needed,

quickly and without compromising existing behavior, is vital. This state machine platform makes this easy, with the abstraction of the state machine organization vital.

One of the pitfalls of many strategies is difficulty in encapsulating and abstracting code behavior. State machines and states allow for this encapsulation and abstraction to occur in a straightforward manner. Each state-machine represents a component of the software, such as a valve, door, driver, engine, etc. that can do different things. Each state of said state machine then represents a mode of behavior the component can operate in. This allows for a natural, mechanical means of separating and abstracting program behavior into easily digestible pieces.

By dividing the program into pieces, the QP framework aids in distributing the burden of the software. Separating program behavior into the board support package and state machines allows for each member of a development team to focus on the aspect with which they are familiar. If a developer has a computer scientist, they would naturally be a good choice for the support package. The state machine controlling a motor, servo, valve, etc. would naturally fall on someone familiar with digital controls and so on. In addition to separating the responsibility for program behavior among team members, state machines allow for distributing the program among hardware platforms as well. The state machine for a motor might communicate with a state machine for a valve. There is no need for the motor and valve to run on the same chip, so long as a means of communication exists between the chips. By running state machines on different chips, the QP framework provides a consistent means of utilizing distributed computing power.

A common limitation of many advanced and flexible coding strategies is ease of understanding. Many of the more powerful and flexible platforms require an extensive and intimate understanding of computer science and programming. Without such an understanding, it is difficult to get desired program behavior, and nearly impossible to understand exactly what is

going on. The QP platform, as the author has attempted to show, functions in a manner similar to real-world mechanical objects. To summarize, first an objects behavior is a function of the state the object is in. Second, an object responds to external events. Third, an object only changes behavior when it changes state. Fourth, an object can be built up of smaller objects, each having states and so forth. A strength of the QP platform is that by organizing program behavior in a manner similar to real-world objects, defining and understanding program behavior becomes much more accessible for a variety of people.

A feature of many modern development processes is the need for testing and simulation. Given the complex nature of many modern products, along with the time and cost involved in creating prototypes, it is useful to test portions of a project or program. By simulating and testing parts of the final product individually, possibly simulating mechanical components in software, significant saving can be realized. This is only possible if the software coding strategy allows it. The QP platform, by working on a large variety of hardware platforms, and encoding most behavior in the state machines, facilitates this. In many cases, the state machines can be coded in such a way that no changes need to be made when moving them between platforms and simulations. Only the board support packages would need to change. This allows for thorough testing of portions of the program behavior piece by piece. Combined with an understanding of how the QP platform functions on each platform, the Arduino Uno being reviewed by the author, allows for detailed analysis. This makes the QP platform excellent for testing, simulation, and prototyping purposes.

The last and perhaps most important feature a software strategy requires for embedded and or real-time systems is reliability. No matter how well a strategy incorporates the previously discussed characteristics, if it is nut utterly reliable, it is useless for many applications. An easy to understand, cheap, lightweight piece of software for a brake servo is worse than useless if it has a 1-in-a-million chance of freezing. Memory leaks and non-deterministic behavior are

common culprits of this. The QP platform does not have this problem. Memory for the events is pre-allocated in a deterministic fashion, preventing lag or leaks. This does mean that sufficient memory must be allocated ahead of time, but that can be determined through testing or math. Likewise, as the data collected in Chapter 4 shows, the platform is highly deterministic. Each given operation in the QP framework takes a precise amount of time. This deterministic nature makes the QP platform usable in real-time applications where performance must meet guaranteed parameters.

While the author is unfamiliar with the exact characteristics of other frameworks and operating systems, the results of the analysis so far has made the author confident of two things. First, the combination of state-machine architecture and the graphical QM tool to precisely and accurately control program behavior would be far simpler to understand and use for those not already intimately familiar with other methods. Second, the processing cost for this simplicity is likely small enough to justify use in even mass produced embedded systems. Without a large, experienced, and thus expensive, team of computer science coders, which would likely require proprietary software tools as well, it is unlikely to get much better performance. Thus, while it may not be the preferred approach for Apple or Microsoft, for companies which are either smaller or not focused almost exclusively on computer programming, it might be worth adopting.

Recommendations

Due to the flexibility of the QP framework, there are many possibilities for further exploration. Perhaps the first that comes to mind is an exploration of the pre-emptive kernel running on the Arduino. A second of course to explore the timing characteristics of the QP-C++ framework on other platforms. A third avenue would be to explore the QP-C or QP-nano versions. A fourth avenue would be to thoroughly explore the memory requirements of the platform, flash, stack, and RAM. The QP framework also includes a powerful internal monitoring service called QSPY

which could be enabled to assist in analysis of a given application. Let us briefly discuss these further.

The QP framework includes both a cooperative and pre-emptive version. For the timing analysis in this thesis we used the cooperative version. Much of the theoretical work for the cooperative version would remain unchanged for a pre-emptive analysis. The main change is that nearly any operation done by or with an Active Object could be interrupted by an operation with a higher priority Active Object. It would be relatively simple, possibly simpler than the cooperative version, to time a basic operation by the highest priority Active Object. One could start a pulse in a lower priority object, create and post an event to the high priority Active Object, then end the pulse. In the pre-emptive kernel, the Active Object with high priority would take over when the event was posted, only returning to the low priority object when finished. This could be chained to explore various combinations. It is likely that interrupting a transition in operation would add ~6-8 cycles to the time it takes to execute, as a function call takes 4 cycles in the Arduino UNO, one call out, one call back to return. The addition of a third Arduino, or alteration of the timer code could also be used to specifically cause interruptions at desired times. As is hopefully apparent, extending the analysis in this paper to the pre-emptive kernel might easily provide the material for another thesis or creative component.

A perhaps obvious avenue for future information gathering efforts would be to run the experiments on other hardware platforms. As long as a board support package can be developed for the desired platform, many of which are already available, the QP framework can run on many target environments, from miniscule microcontrollers to massive supercomputers. It is likely that there would be variations in the number of cycles which would be required for different tasks. This variation can come from the way compilers create the code, formatting and configuration used in porting the platform, as well as the way in which different chips run the

instructions. As such, before using the QP framework in a project, it might be prudent to run the timing analysis so that performance characteristics can be known precisely.

While the C++ version of the QP framework was used in this thesis due to an already existing port to the Arduino UNO, there are two other versions. As the author understands it the C version is essentially identical, with the C++ version merely a translation in program semantics. The C version is mainly provided because some C++ compilers for microcontrollers produce code with serious performance issues. The nano version of QP is quite different. Many features are removed or restricted. This is because the primary purpose of the QP nano version is to create a framework which can run state machines with minimal resources. The creator of the QP framework specifically stated as a goal for QP-nano, a feasible framework to be used by the individual components, threads, and cores of a chip to aid in the development of microcontrollers and processors. As such the QP-nano, in particular, could serve as something to further explore.

One important characteristic of any program framework is memory. Program memory, stack space, and RAM are precious resources in many embedded systems. As such, a detailed understanding of how and how much of each is used could prove useful. In this thesis, the focus was on understanding the state machine architecture and the timing performance. Developing an analysis of memory usage, would likely be a worthy pursuit. Speed, reliability, and memory usage are vital characteristics. The strongly deterministic nature of the timing has already been shown. With an analysis of memory, a developer looking for a strategy would then have a good idea of what they would get from QP.

Of course the final test of any strategy is how it functions in application. With an understanding of timing and memory with the platform in use, it remains necessary to understand the specific application. The QP framework includes a monitoring service built into the code called QSPY. This service when enabled in the compiler, provides many services for monitoring the framework

as well as the behavior of the components such as the state machines, event queues, memory pools, etc. This would allow a developer to get a view of what is going on under the hood that might not otherwise be possible. As for why this was not explored in this thesis, there are two main reasons. First was that while intended to be lightweight, it would inevitably alter the timing characteristics slightly. Second, the timing functions QSPY provides have a resolution of 256 microseconds on the Arduino Uno. This was far too coarse of a resolution for distinguishing the time various framework tasks take. It might however be useful for practical applications, especially since it does preserve the order in which things are logged. For a fairly complex set of Active Objects, activating the QSPY service in various debug versions, or even production versions if the ability to monitor justifies the resource usage, could conceivably greatly aid in nearly any application.

Important for many who would explore it as well is the fact that it is free to learn and use, an advantage over many less widely applicable alternatives. The author has hope that the reader has been convinced that the QP framework is rich in opportunity. First, it has a wide range of applicability in practical applications. Second, it presents a near unlimited number of avenues for learning or aiding development.

REFERENCES

Atmel Corporation. (2012, August). *ATmega328P Data Sheet.* Retrieved November 27, 2012, from Atmel Home- Atmel Corporation: http://www.atmel.com/Images/doc8271.pdf

Auslander, D. M., Ridgely, J. R., & Ringgenberg, J. D. (2002). *Control Software for Mechanical Systems: Object-Oriented Design in a Real-Time World.* Upper Saddle River, NJ: Prentice Hall PTR.

Bartos, F. J. (2007). Simulation widens mechatronics. *Control Engineering*, 26.

Costello, S. (2012, October 25). *How Many iPhones Have Been Sold Worldwide?* Retrieved October 25, 2012, from About.com: http://ipod.about.com/od/glossary/f/how-many-iphones-sold.htm

Li, Q., & Yao, C. (2003). *Real-time concepts for embedded systems.* San Francisco: CMP Books.

Microchip Technology Inc. (2012, October 22). *Microcontroller Product Selector (MPS)* . Retrieved October 22, 2012, from Microchip Technology Inc. Web Site: http://www.microchip.com/productselector/MCUProductSelector.html

Microchip Technology Incorporated. (2008, November 21). *PIC18F2420/2520/4420/4520 Data Sheet.* Retrieved November 27, 2012, from Microchip.com: http://ww1.microchip.com/downloads/en/DeviceDoc/39631E.pdf

Murray, C. J. (2009, July 29). *Design News - Features - Automakers Aim to Simplify Electrical Architectures*. Retrieved November 2, 2012, from Design News - Serving the 21st Century Design Engineer: http://www.designnews.com/document.asp?doc_id=228519

National Instruments. (2012, August 15). *Do I Need a Real-Time System?* Retrieved November 13, 2012, from National Instruments: Test, Measurement, and Embedded Systems: http://www.ni.com/white-paper/14238/en

National Instruments. (2012, May 21). *What is a Real-Time Operating System (RTOS)? - Developer Zone - National Instruments.* Retrieved June 6, 2012, from National Instruments: Test, Measurement, and Embedded Systems: http://www.ni.com/white-paper/3938/en

Object Management Group. (2011, August 06). *UML 2.4.1.* Retrieved March 16, 2013, from Object Management Group Web site: http://www.omg.org/spec/UML/2.4.1/

Rosenthal, S. (1995, May). *Interrupts might seem basic, but many programmers still avoid them.* Retrieved December 2, 2012, from SLTF Consulting - Firmware Developed For You: http://www.sltf.com/articles/pein/pein9505.htm

Rumbaugh, J. R., Blaha, M. R., Lorenson, W., Eddy, F., & Premerlani, W. (1990). *Object-Oriented Modeling and Design.* Prentice-Hall.

Samek, M. (2008). *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems.* Burlington, MA: Newnes.

Walls, C. (2010, January 11). *Deterministic dynamic memory allocation & fragmentation in C & C++*. Retrieved October 25, 2012, from Embedded: http://www.embedded.com/design/prototyping-and-development/4008868/Deterministic-dynamic-memory-allocation--fragmentation-in-C--C-?page=0

Wikimedia Foundation, Inc. (2012, September 23). *List of common microcontrollers*. Retrieved October 22, 2012, from Wikipedia,the free encyclopedia: http://en.wikipedia.org/wiki/List_of_common_microcontrollers

Wright, D. R. (2005). *CSC215 Class Notes.* Retrieved october 23, 2012, from Prof. David R. Wright website: http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf

APPENDICES

Appendix A: Code

NSTimer.ino (Arduino project file)

```
#include "qp_port.h"
#include "NSTimer.h"
#include "bsp.h"
#include "Arduino.h" // always include in your sketch

Q_DEFINE_THIS_FILE

using namespace QP;
// Local-scope objects -----------------------------------------------------
static QEvt const *NSTimer_queueSto[10]; // alocate event queue buffer

static QF_MPOOL_EL(QEvt) l_smlPoolSto[10]; // storage for the small event pool

//............................................................................
void setup() {
        // initialize the BSP
        BSP_init();
        // initialize the framework and the underlying RT kernel
        QF::init();
        // initialize event pools...
        QF::poolInit(l_smlPoolSto, sizeof(l_smlPoolSto), sizeof(l_smlPoolSto[0]));
        AO_NSTimer->start(1, NSTimer_queueSto, Q_DIM(NSTimer_queueSto),
                (void *)0, 0U); // start the NSTimer active object
}
```

NSTimer.h

```
#ifndef NSTimer_h
#define NSTimer_h
using namespace QP;

enum NSTimerSignals { // signals for the NSTimer application
        TIMEOUT_SIG = Q_USER_SIG,
        A_SIG,
        B_SIG,
        C_SIG,
        D_SIG,
        E_SIG,
        F_SIG,
        G_SIG,
        H_SIG
};

// active objects ..................................................
$declare(components::AO_NSTimer) // "opaque" pointer to NSTimer AO

#endif // NSTimer_h
```

bsp.h

```
#ifndef bsp_h
#define bsp_h

#include <avr/io.h> // AVR I/O

// Sys timer tick per seconds
#define BSP_TICKS_PER_SEC       100
#define PULSE_ON()              (PORTD |= (1 << (2)))
#define PULSE_OFF()             (PORTD &= ~(1 << (2)))

void BSP_init(void);
void BSP_ledOff(void);
void BSP_ledOn(void);
#endif // bsp_h
```

bsp.cpp

```
#include "qp_port.h"
#include "NSTimer.h"
#include "bsp.h"
```

```
#include "Arduino.h"              // Arduino include file

Q_DEFINE_THIS_FILE

int flag=0;
//............................................................................
void BSP_init(void) {
        DDRB = 0xFF; // All PORTB pins are outputs (user LED)
        PORTB = 0x00; // drive all pins low
}

//............................................................................
void BSP_ledOff(void) {
        PORTB &= ~(1 << 5);
}

//............................................................................
void BSP_ledOn(void) {
        PORTB |= (1 << 5);
}

//............................................................................
void QF::onStartup(void) {
}

//............................................................................
void QF::onCleanup(void) {
}

//............................................................................
void QF::onIdle() {
        //We are idle, so we are done processing
        PULSE_OFF();
        //Flip flag on and off for test/reset
        flag!=1;
        //test if flag off
        if(!flag)
        {
        //Make sure timer board has time to prepare
                delay(15);
        //Start Pulse
                PULSE_ON();
        //Post event signal A to start transition
                AO_NSTimer->POST(Q_NEW(QEvt, A_SIG), &onIdle);
        }
        //Return to Origin if flag is on with signal B
```

```
	else
	{
		AO_NSTimer->POST(Q_NEW(QEvt, B_SIG), &onIdle);
	}
}

//............................................................................
void Q_onAssert(char const Q_ROM * const Q_ROM_VAR file, int line) {
	QF_INT_DISABLE();           // disable all interrupts
	BSP_ledOn();                // User LED permanently ON
	asm volatile ("jmp 0x0000"); // perform a software reset of the Arduino
}
```

ao_NSTimer.cpp

```
#include "qp_port.h"
#include "bsp.h"
#include "NSTimer.h"
#include "Arduino.h"
//Q_DEFINE_THIS_FILE
// NSTimer class -------------------------------------------------------------
$declare(components::NSTimer)
// Local objects -------------------------------------------------------------
static NSTimer l_NSTimer; // the single instance of NSTimer active object
// Global objects ------------------------------------------------------------
QActive * const AO_NSTimer = &l_NSTimer; // the opaque pointer
// Pelican class definition --------------------------------------------------
$define(components::NSTimer)
```

Appendix B: Nesting Level 3 Data

Min(us)

Min (us)		Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	56	45	45	45	56	56
	s1 - IT to s11	73	70	48	70	73	69
	s1 - IT to s111	81	81	82	77	77	81
	s1 - IT to s11 - IT to s111	87	86	88	82	83	87
	s11	61	56	48	56	61	57
	s11 - IT to s111	77	73	70	69	73	77
	s111	75	61	56	57	70	75
	s112	74	61	49	61	75	75
	s12	61	48	48	48	61	61
	s12 - IT to s121	77	50	50	50	77	77
	s121	75	49	49	49	75	74
	s2	45	45	45	45	45	45
	s2 - IT to s21	48	48	48	48	48	48
	s2 - IT to s21 - IT to s211	50	50	50	50	50	50
	s2 - IT to s211	50	50	50	50	50	50
	s21 - IT to s211	50	50	50	50	50	50
	s21	48	48	48	48	48	48
	s211	49	49	49	49	49	49
	Internal	44	44	44	45	48	45

Mean(us)

Mean (us)		Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	56.900	46.007	46.012	46.009	56.977	56.959
	s1 - IT to s11	73.131	70.082	49.005	70.197	73.125	69.036
	s1 - IT to s111	81.190	81.455	82.348	77.352	77.075	81.184
	s1 - IT to s11 - IT to s111	87.905	86.763	88.799	82.677	84.636	87.908
	s11	61.966	56.873	48.378	56.956	61.957	57.863
	s11 - IT to s111	77.651	73.135	70.077	69.030	73.548	77.655
	s111	75.026	61.985	56.879	57.863	70.976	75.036
	s112	75.033	61.970	49.757	61.967	75.028	75.032
	s12	61.965	48.363	48.349	48.319	61.970	61.963
	s12 - IT to s121	77.684	50.737	50.703	50.667	77.635	77.658
	s121	75.032	49.778	49.781	49.750	75.026	75.023
	s2	46.005	46.015	46.010	46.006	46.011	46.001
	s2 - IT to s21	49.002	49.006	49.004	49.003	49.003	48.998
	s2 - IT to s21 - IT to s211	51.993	52.000	51.996	51.997	51.997	51.999
	s2 - IT to s211	50.719	50.693	50.717	50.683	50.667	50.719
	s21 - IT to s211	50.621	50.689	50.725	50.691	50.693	50.667
	s21	48.354	48.356	48.361	48.342	48.346	48.320
	s211	49.756	49.790	49.766	49.769	49.775	49.764
	Internal	44.510	44.515	44.513	45.917	48.243	45.923

Max(us)

Max (us)		Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	58	48	48	48	58	58
	s1 - IT to s11	74	71	50	71	74	70
	s1 - IT to s111	82	82	83	78	78	82
	s1 - IT to s11 - IT to s111	88	87	90	83	85	88
	s11	64	57	49	58	64	60
	s11 - IT to s111	78	74	71	70	74	78
	s111	77	64	57	58	73	77
	s112	77	64	50	64	77	77
	s12	62	49	49	49	64	64
	s12 - IT to s121	78	52	52	52	78	78
	s121	77	50	50	50	77	77
	s2	48	48	48	48	48	48
	s2 - IT to s21	50	50	50	50	50	50
	s2 - IT to s21 - IT to s211	53	53	53	53	53	53
	s2 - IT to s211	52	52	52	52	52	52
	s21 - IT to s211	52	52	52	52	52	52
	s21	49	49	49	49	49	49
	s211	50	50	50	52	50	50
	Internal	45	45	45	48	49	48

Floor(cycles)

Floor (cycles)		Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	224	180	180	180	224	224
	s1 - IT to s11	292	280	192	280	292	276
	s1 - IT to s111	324	324	328	308	308	324
	s1 - IT to s11 - IT to s111	348	344	352	328	332	348
	s11	244	224	192	224	244	228
	s11 - IT to s111	308	292	280	276	292	308
	s111	300	244	224	228	280	300
	s112	296	244	196	244	300	300
	s12	244	192	192	192	244	244
	s12 - IT to s121	308	200	200	200	308	308
	s121	300	196	196	196	300	296
	s2	180	180	180	180	180	180
	s2 - IT to s21	192	192	192	192	192	192
	s2 - IT to s21 - IT to s211	200	200	200	200	200	200
	s2 - IT to s211	200	200	200	200	200	200
	s21 - IT to s211	200	200	200	200	200	200
	s21	192	192	192	192	192	192
	s211	196	196	196	196	196	196
	Internal	176	176	176	180	192	180

Mean(cycles)

Mean (cycles)		Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	227.60	184.03	184.05	184.04	227.91	227.84
	s1 - IT to s11	292.52	280.33	196.02	280.79	292.50	276.14
	s1 - IT to s111	324.76	325.82	329.39	309.41	308.30	324.74
	s1 - IT to s11 - IT to s111	351.62	347.05	355.20	330.71	338.54	351.63
	s11	247.86	227.49	193.51	227.82	247.83	231.45
	s11 - IT to s111	310.60	292.54	280.31	276.12	294.19	310.62
	s111	300.10	247.94	227.52	231.45	283.90	300.14
	s112	300.13	247.88	199.03	247.87	300.11	300.13
	s12	247.86	193.45	193.40	193.28	247.88	247.85
	s12 - IT to s121	310.74	202.95	202.81	202.67	310.54	310.63
	s121	300.13	199.11	199.12	199.00	300.10	300.09
	s2	184.02	184.06	184.04	184.02	184.04	184.00
	s2 - IT to s21	196.01	196.02	196.02	196.01	196.01	195.99
	s2 - IT to s21 - IT to s211	207.97	208.00	207.98	207.99	207.99	208.00
	s2 - IT to s211	202.88	202.77	202.87	202.73	202.67	202.88
	s21 - IT to s211	202.48	202.76	202.90	202.76	202.77	202.67
	s21	193.42	193.42	193.44	193.37	193.38	193.28
	s211	199.02	199.16	199.06	199.08	199.10	199.06
	Internal	178.04	178.06	178.05	183.67	192.97	183.69

Ceiling(cycles)

Ceiling (cycles)		Transition Origins					
		s1(s1)	s11(s11)	s111(s111)	s11(s111)	s1(s111)	s1(s11)
Transition Destinations	s1	232	192	192	192	232	232
	s1 - IT to s11	296	284	200	284	296	280
	s1 - IT to s111	328	328	332	312	312	328
	s1 - IT to s11 - IT to s111	352	348	360	332	340	352
	s11	256	228	196	232	256	240
	s11 - IT to s111	312	296	284	280	296	312
	s111	308	256	228	232	292	308
	s112	308	256	200	256	308	308
	s12	248	196	196	196	256	256
	s12 - IT to s121	312	208	208	208	312	312
	s121	308	200	200	200	308	308
	s2	192	192	192	192	192	192
	s2 - IT to s21	200	200	200	200	200	200
	s2 - IT to s21 - IT to s211	212	212	212	212	212	212
	s2 - IT to s211	208	208	208	208	208	208
	s21 - IT to s211	208	208	208	208	208	208
	s21	196	196	196	196	196	196
	s211	200	200	200	208	200	200
	Internal	180	180	180	192	196	192

Appendix C: Lists of Origins and Destinations

Nesting Level 1

Transition Origins
s1(s1)

Non-Parallel ITs
None

Basic
s1
s2
Internal

Parallel Substates
None

Parallel ITs
None

Nesting Level 2

Transition Origins
s1(s1)
s11(s11)
s1(s11)

Non-Parallel ITs
s1 - IT to s11
s2 - IT to s21

Basic
s1
s11
s2
s21
Internal

Parallel Substates
s12

Parallel ITs
None

Nesting Level 3

Transition Origins
s1(s1)
s11(s11)
s111(s111)
s11(s111)
s1(s111)
s1(s11)

Basic
s1
s11
s111
s2
s21
s211
Internal

Non-Parallel ITs
s1 - IT to s11
s1 - IT to s111
s1 - IT to s11 - IT to s111
s11 - IT to s111
s2 - IT to s21
s2 - IT to s21 - IT to s211
s2 - IT to s211
s21 - IT to s211

Parallel Substates
s112
s12
s121

Parallel ITs
s12 - IT to s121

Nesting Level 4

Transition Origins
s1(s1)
s11(s11)
s111(s111)
s1111(s1111)
s111(s1111)
s11(s1111)
s1(s1111)
s11(s111)
s1(s111)
s1(s11)

Basic
s1
s11
s111
s1111
s2
s21
s211
s2111
Internal

Parallel Substates
s1112
s112
s1121
s12
s121
s1211

Non-Parallel ITs
s1 - IT to s11
s1 - IT to s111
s1 - IT to s1111
s1 - IT to s11 - IT to s111
s1 - IT to s11 - IT to s1111
s1 - IT to s111 - IT to s1111
s1 - IT to s11 - IT to s111 - IT to s1111
s11 - IT to s111
s11 - IT to s1111
s11 - IT to s111 - IT to s1111
s111 - IT to s1111
s2 - IT to s21
s2 - IT to s211
s2 - IT to s2111
s2 - IT to s21 - IT to s211
s2 - IT to s21 - IT to s2111
s2 - IT to s211 - IT to s2111
s2 - IT to s21 - IT to s211 - IT to s2111
s21 - IT to s211
s21 - IT to s2111
s21 - IT to s211 - IT to s2111
s211 - IT to s2111

Parallel ITs
s12 - IT to s121
s12 - IT to s1211
s12 - IT to s121 - IT to s1211
s121 - IT to s1211
s112 - IT to s1121

VITA

Christopher Weathers

Candidate for the Degree of

Master of Science

Thesis: REVIEW AND TIMING ANALYSIS OF THE REAL TIME QP STATE MACHINE FRAMEWORK

Major Field: Mechanical and Aerospace Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in your Mechanical and Aerospace Engineering at Oklahoma State University, Stillwater, Oklahoma in July, 2014.

Completed the requirements for the Bachelor of Science in Mechanical and Aerospace Engineering at Oklahoma State University, Stillwater, Oklahoma in May 2009.

Experience:

Teaching Assistant from Fall 2009-Spring 2012

Professional Memberships:

Pi Tau Sigma

Tau Beta Pi

Phi Kappa Phi